

Redundancy, Context, and Preference: An Empirical Study of Duplicate Pull Requests in OSS Projects

Zhixing Li, Yue Yu*, Minghui Zhou, Tao Wang, Gang Yin, Long Lan, and Huaimin Wang

Abstract—OSS projects are being developed by globally distributed contributors, who often collaborate through the pull-based model today. While this model lowers the barrier to entry for OSS developers by synthesizing, automating and optimizing the contribution process, coordination among an increasing number of contributors remains as a challenge due to the asynchronous and self-organized nature of distributed development. In particular, duplicate contributions, where multiple different contributors unintentionally submit duplicate pull requests to achieve the same goal, are an elusive problem that may waste effort in automated testing, code review and software maintenance. While the issue of duplicate pull requests has been highlighted, to what extent duplicate pull requests affect the development in OSS communities has not been well investigated. In this paper, we conduct a mixed-approach study to bridge this gap. Based on a comprehensive dataset constructed from 26 popular GitHub projects, we obtain the following findings: (a) Duplicate pull requests result in redundant human and computing resources, exerting a significant impact on the contribution and evaluation process. (b) Contributors' inappropriate working patterns and the drawbacks of their collaborating environment might result in duplicate pull requests. (c) Compared to non-duplicate pull requests, duplicate pull requests have significantly different features, *e.g.*, being submitted by inexperienced contributors, being fixing bugs, touching cold files, and solving tracked issues. (d) Integrators choosing between duplicate pull requests prefer to accept those with early submission time, accurate and high-quality implementation, broad coverage, test code, high maturity, deep discussion, and active response. Finally, actionable suggestions and implications are proposed for OSS practitioners.

Index Terms—Duplicate pull requests, pull-based development model, distributed collaboration, social coding



1 INTRODUCTION

The success of many community-based Open Source Software (OSS) projects relies heavily on a large number of volunteer developers [35], [64], [84], [86], who are geographically distributed and collaborate online with others from all over the world [43], [58]. Compared to the traditional email-based contribution submission [25], the pull-based model [40] on modern collaborative coding platforms (*e.g.*, GitHub [9] and GitLab [10]) supports a more efficient collaboration process [115], by coupling code repository with issue tracking, review discussion and continuous integration, delivery and deployment [79], [110]. Consequently, an increasing number of OSS projects are adopting the synthesized pull-based mechanism, which helps them improve their productivity [98] and attract more contributors [108].

However, while the increased number of contributors in large-scale software development leads to more innovations

(*e.g.*, unique ideas and inspiring solutions), it also results in severe coordination challenges [103]. Currently, one of the typical coordination problems in pull-based development is duplicate work [85], [114], due to the asynchronous nature of loosely self-organized collaboration [26], [84] in OSS communities. On the one hand, it is unreasonable for a core team to arrange and assign external contributors to carry out every specific task under the open source model [53], [54] (*i.e.*, external contributors are mainly motivated by interest and intellectual stimulation derived from writing code, rather than requirements or assignments). On the other hand, it is impractical to expect external developers (especially newcomers and occasional contributors) to deeply understand the development progress of the OSS projects [41], [56], [83] before submitting patches. Thus, OSS developers involved in the pull-based model submit *duplicate pull requests* (akin to duplicate bug reports [24]), even though they collaborate on modern social coding platforms (*e.g.*, GitHub) with relatively transparent [37], [94] and centralized [40] working environments. The recent study by Zhou *et al.* [114] has showed that complete or partial duplication is pervasive in OSS projects and particularly severe in some large projects (max 51%, mean 3.4%).

Notably, a large part of duplicates are not submitted intentionally to provide different or better solutions. Instead, contributors submit duplicates unintentionally because of misinformation and unawareness of a project's status [41], [114]. In practice, duplicate pull requests may cause substantial friction among external contributors and core inte-

- Zhixing Li, Yue Yu, Tao Wang, Gang Yin and Huaimin Wang are with the Key Laboratory of Parallel and Distributed Computing, College of Computer, National University of Defense Technology, Changsha, China. E-mail: {lizhixing15, yuyue, taowang2005, yingang, hmwang}@nudt.edu.cn
- Minghui Zhou is with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. E-mail: zhmh@pku.edu.cn
- Long Lan, Minghui Zhou and Yue Yu are with the Peng Cheng Laboratory, Shenzhen, China. E-mail: long.lan@pcl.ac.cn

*Corresponding author: Yue Yu, yuyue@nudt.edu.cn

grators; these duplicates are a common reason for direct rejection [41], [85] without any chance for improvement, which frustrates contributors and discourages them from contributing further. Moreover, redundant work is more likely to increase costs during the evaluation and maintenance stages assembled with DevOps tools compared to traditional development models. For example, continuous integration tools (Travis-CI [98], [104]) automatically merge every newly received pull request into a testing branch, build the project and run existing test suites, so computing resources are wasted if integrators do not discover the duplicates and stop the automation process in time. Therefore, avoiding duplicate pull requests is becoming a realistic demand for OSS management, *e.g.*, scikit-learn provides a special note in the contributing guideline *“To avoid duplicating work, it is highly advised that you search through the issue tracker and the PR list. If in doubt about duplicated work, or if you want to work on a non-trivial feature, it is recommended to first open an issue in the issue tracker to get some feedbacks from core developers.”* [5]

Existing work has highlighted the problems of duplicate pull requests [40], [85], [114] (*e.g.*, inefficiency and redundant development), and proposed ways to detect duplicates [57], [73]. However, the nature of duplicate pull requests, particularly the fine-grained resources that are wasted by the duplicates, the context in which duplicates occur, and the features that distinguish merged duplicates from their counterparts, have rarely been investigated. Understanding these questions would help mitigate the threats brought by duplicate pull requests and improve software productivity.

Therefore, we bridge the gap on the investigation of duplicate pull requests in this study. We extend our previously collected duplicate pull request dataset [106] by adding change details, review history, and integrators’ choice. Based on the dataset, we analyze the redundancies of duplicate pull requests in the development and evaluation stages, explore the context in which duplicates occur and examine the difference between duplicate and non-duplicate pull requests. We further investigate the reasons why among a group of duplicates, a pull request is more likely to be accepted by an integrator. Finally, we propose actionable suggestions for OSS communities.

The main contributions of this paper are summarized as follows:

- It presents empirical evidence on the impact of duplicate pull requests on development effort and review process. The findings will help software engineering researchers and practitioners better understand the threats of duplicate pull requests.
- It reveals the context of duplicate pull requests, highlighting the inappropriateness of OSS contributors’ work patterns and the shortcomings of the current OSS collaboration environment. These findings can guide developers to avoid redundant effort on the same task.
- It provides quantitative insights into the difference between duplicate and non-duplicate pull requests, which can offer useful guidance for automatic duplicate detection.
- It summarizes the characteristics of the accepted pull requests compared to those of their duplicate coun-

terparts, which will provide actionable suggestions for inexperienced integrators in duplicate selection.

The rest of the paper is organized as follows: Section 2 introduces the background and research questions. Section 3 presents the dataset used in this study. Sections 4, 5 and 6 report the experimental results and findings. Section 7 provides further discussion and proposes actionable suggestions and implications for OSS practitioners. Section 8 discusses the threats to the validity of the study. Finally, we draw conclusions in Section 9.

2 BACKGROUND AND RESEARCH QUESTIONS

2.1 Pull-based development

In the global collaboration of OSS projects, a variety of tools [115], including mailing lists, bug trackers (*e.g.*, Bugzilla [3]), and source code version control systems (*e.g.*, SVN and Git), have been widely used to facilitate collaboration processes. The pull-based development model is the latest paradigm [40] for distributed development; it integrates code base with task management, code review and DevOps toolset. Compared with the traditional patch-based model, the pull-based model provides OSS developers with centralization of information, integration of tools, and process automation, thus simplifying the participation process and lowering the entry barrier for contributors [40], [98]. In addition, the pull-based model separates the developers into two teams, *i.e.*, the external contributor team, which does not have the write access to the repository and submits contributions via pull requests, and the core integrator team, which is responsible for assessing and integrating the pull requests sent from external contributors. This decoupling of effort stimulates and enhances the parallel and distributed collaboration among OSS developers [115]. As shown in Figure 1, the pull-based development workflow [31] includes the following steps.

a) *Fork*: For a contributor (*Bob* or *alice*) in GitHub, the first step to contribute to a project is forking its original repository. As a result, the contributor owns a copy of the repository containing all the source code and commit histories under her/his GitHub account. Both the original repository and the forked repository are hosted on the servers of GitHub.

b) *Clone*: Before the contributor engages in actual work, s/he must clone the forked repository to her/his local computer and makes code changes based on the local repository.

c) *Edit*: The contributor can then fix bugs or add new features by editing the local repositories. Moreover, the contributor is recommended to always create a *topic branch* separated from the *master branch* and to commit local changes to that topic branch.

d) *Sync*: It is possible that the local repository becomes out of date compared with the original repository. To make it easy for project integrators to merge the local changes cleanly, the contributor is expected first to sync the latest commits from the original repository and handle the possible merge conflicts.

e) *Push*: The contributor then pushes the local changes to the forked repository. The forked repository acts as a transfer station of the local changes from the local repository to the original repository.

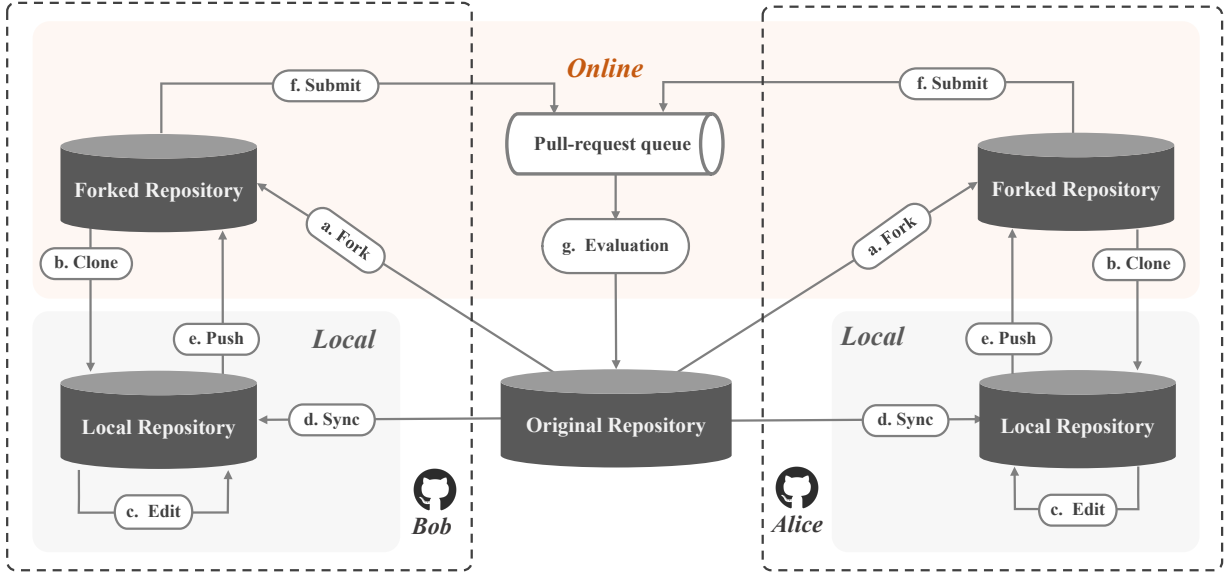


Fig. 1: The workflow of the pull-based development model.

f) *Submit*: Based on the forked repository, the contributor issues a pull request to notify the project integrators to merge (*i.e.*, pull) the pushed commits. The pull request consists of a title and a description, which are used as a straightforward elaboration of the contained commits. All pull requests submitted to a specific project are maintained in a queue (*i.e.* the issue tracker), and each developer can check the status and review histories of the pull requests.

g) *Evaluation*: To ensure that the submitted changes do not contain defects and adhere to project conventions, the project integrators and other community developers who are interested in the project discuss the appropriateness and quality of the pushed changes. Finally, the integrators reach an agreement on whether to accept the changes after several rounds of discussion.

2.2 Evaluation and decision of OSS code contributions

The evaluation and decision of OSS contributions is a comprehensive process [93], [98], [107], in which the code reviewers would consider various factors. Previous studies have attempted to use quantitative methods to uncover the characteristics of accepted contributions or use qualitative methods to explore the factors that integrators examine when making decisions. Those studies provide a good guidance on which factors should be considered as controls when we analyze the characteristics of and integrators' preference among duplicate pull requests.

Characteristics of accepted contributions. Rigby *et al.* [76] analyzed the patches submitted to the Apache server project, and found that patches of small size are more likely to be accepted than large ones in that project. The similar finding was also reported by Weissgerber *et al.* [101] in their study on two other OSS projects. Jiang *et al.* [47] conducted a case study on the Linux kernel project, which showed that patches submitted by experienced developers, patches of high maturity, and patches changing popular subsystems are more likely to be accepted. Baysal *et al.* [20] also found

that developer's experience has a positive effect on patch acceptance in the WebKit and Google Blink projects. In the pull request model specifically, Gousios *et al.* [40] found that the hotness of project area is the dominating factor affecting pull request acceptance. Tsay *et al.* [93] investigated the effects of both technical and social factors on pull request acceptance. Their findings showed that stronger social connection between the pull request author and integrators can increase the likelihood of pull request acceptance. Yu *et al.* [108] investigated pull request evaluation in the context of CI (continuous integration). They found that CI testing results significantly influence the outcome of pull request review and the pull requests failing CI tests have a high likelihood of rejection. Kononenko *et al.* [52] studied the pull requests submitted to a successful commercial project. Their analysis results presented that patch size, discussion length, and authors' experience and affiliation are important factors affecting pull request acceptance in that project. A recent study conducted by Zou *et al.* [116] showed that pull requests with larger code style inconsistency are more likely to be rejected.

Factors that integrators examine when making decisions. Rigby *et al.* [77] interviewed with nine core developers from the Apache server project on why they rejected a patch. Although technical issue is the dominating reason, the reported reasons also include project scope and other political issues. Pham *et al.* [68] interviewed with project owners from GitHub, and found that many factors are considered by project owners when evaluating contributions, *e.g.*, the trustworthiness of contributor and the size, type and target of changes. Tsay *et al.* [94] analyzed integrators' comments and decisions on highly discussed pull requests. They found that integrators are usually polite to new contributors for social encouragement, and integrators' decisions can be affected by community support. Gousios *et al.* [42] surveyed hundreds of integrators from GitHub on how they decide whether to accept a pull request. Their survey results

showed that the most frequently mentioned factors are contribution quality, adherence to project norm, and testing results. Tao *et al.* [89] analyzed the rejected patches from Eclipse and Mozilla, and derived a list of reasons for patch rejection, *e.g.*, compilation errors, test failures, and incomplete fix. Kononenko *et al.* [51] surveyed core developers from the project Mozilla and asked them about the top factors affecting the decision of code review. They found that the experience of developers receives the overwhelming number of positive answers. In a recent study, Ford *et al.* [39] investigated the pull request review process on the basis of an eye-tracker dataset collected from direct observation of reviewers' decision making. Interestingly, they found that developers reviewing code contributions in GitHub actually examined the social signals from profile pages (*e.g.*, avatar image) more than they reported.

2.3 Duplicates in community-based collaboration

Duplicate efforts, including duplicate bug reports, duplicate questions, and, recently, duplicate pull requests have been studied in the literature. Studies have mainly focused on revealing the threats in duplicates, and proposed methods to detect and remove duplicates.

Duplicate bug reports. Many OSS projects incorporate bug trackers so that developers and users can report the bugs they have encountered [78], [100]. From Bugzilla to GitHub issue system [2], bug tracking systems have become more lightweight, which makes it easier to submit bug reports. Consequently, popular OSS projects can receive hundreds of bug reports from the community every day [87]. However, because the reporting process is uncoordinated, some bugs might be reported multiple times by different developers. For example, duplicate issues account for 36% and 24% of all reported issues in Gnome and Mozilla [112], respectively, and consume considerable effort from developers to confirm. Meanwhile, researchers have proposed various methods to automatically detect duplicate bug reports. Most of them used similarity-based methods that compute the similarity between a given bug report and previously submitted reports based on various information, including natural language text [78], [100], execution trace [100], categorical features of bugs [87]. Other work has used a machine learning-based classifier [55], [88] and a topic-based model [65] to improve detection performance. It is also interesting to observe that duplicates may attract less attention and consume less effort. For example, Zhou and Mockus found that the issues resolved with FIXED tend to have more comments than other issues, and issues with resolution DUPLICATE tend to have the least comments [112].

Duplicate questions. Stack Overflow [13] is currently the most popular programming Q&A site where developers ask and answer questions related to software development and maintenance [91]. In Stack Overflow, developers can vote on the quality of any question and answer, and they can gain reputation for valued contributions. Since its founding in 2008, Stack Overflow has accumulated 20 million questions and answers and attracted millions of visitors each month. Because of the large user base, Stack Overflow also faces the challenge of receiving duplicate questions posted by

different developers, despite of its explicit suggestion that developers conduct a search first before posting a question. Prior studies have investigated the detection of duplicate questions in Stack Overflow. Zhang *et al.* [111] computed the similarity between two questions based on the titles, descriptions and tags of the questions and latent topic distributions, and they recommended the most similar questions for a given question. To improve detection accuracy, Mizobuchi *et al.* [62] used word embedding to overcome the problem of word ambiguities and catch up new technical words. Zhang *et al.* [109] leveraged continuous word vectors, topic model features and frequent phrases pairs to capture semantic similarities between questions. Moreover, Mizobuchi *et al.* [17] investigated why duplicate questions are submitted and found that not searching for the questions is the most frequent reason. However, submitting questions is significantly different from submitting pull requests in both the form and the process.

Duplicate pull requests. Figure 2 shows an example of a pair of duplicate pull requests, both of which replace function `empty?` with `any?` for better readability. Duplicate pull requests both go through the normal evaluation process until their duplicate relation is identified by the reviewers. Prior studies have reported that duplicate pull requests are a pervasive and severe problem that affects development efficiency [85], [114]. Gousios *et al.* [40] found that more than half of sampling pull requests were rejected due to non-technical reasons and duplication is one of the major problems. Steinmacher *et al.* [85] conducted a survey with the quasi-contributors to obtain their perspectives on reasons for pull request nonacceptance, and duplication was the most common reason mentioned by the quasi-contributors. Furthermore, to help reviewers find duplicate pull requests in a timely manner, researchers [57], [73] have proposed methods to automatically recommend similar pull requests. In addition, Zhou *et al.* [114] explored the weak evidence that discussing or claiming an issue before submitting a pull request correlates with a lower risk of duplicate work. In brief, the above studies have revealed the threats in duplicate pull requests and proposed automatic detection methods for duplicates, but to what extent duplicate pull requests affect the OSS development, the context in which duplicates occur and integrators' choice between duplicates remain unclear.

2.4 Goals and research questions

In this study, we aim to better understand the mechanism of distributed collaboration with pull requests, and to avoid duplication and redundancy in an actionable and effective way. In particular, the main goals of the paper are as follows.

Reveal the impact of duplicates. We aim to obtain quantitative evidence of the impacts of duplicate pull requests on the contribution and evaluation process to more clearly reveal the inefficiency of redundant development.

Guide OSS practitioners. We hope our study can guide OSS contributors to improve their work patterns and avoid unintentional redundant efforts on the same task. Moreover, we expect to help integrators learn from the practices in dealing with duplicates and make more informed decisions about choosing between duplicates.

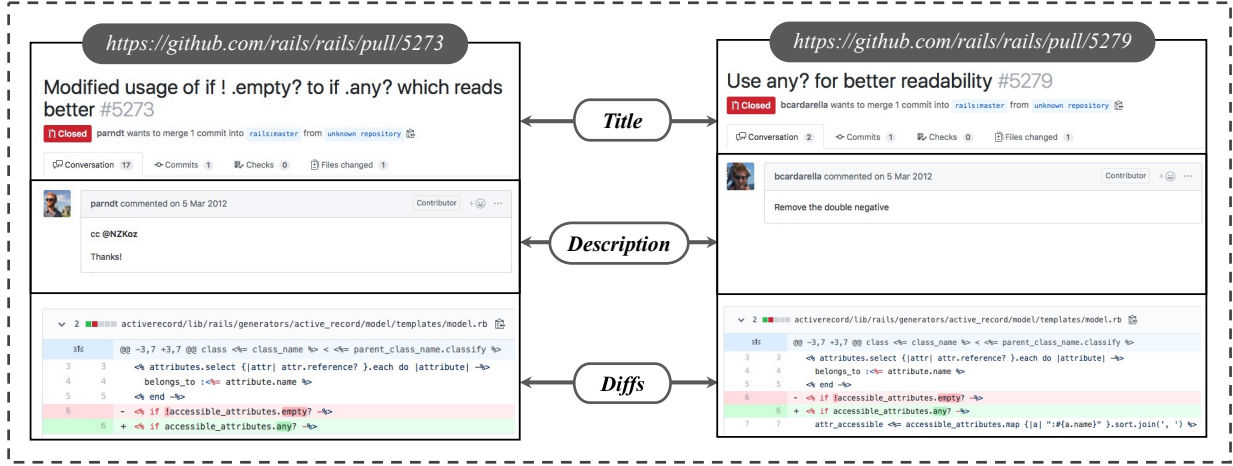


Fig. 2: An example of a pair of duplicate pull requests.

Inspire tool design. We also hope our findings can inspire the OSS community and researchers and provide some insight into how to design and develop mechanisms and tools to assist developers in avoiding, detecting, managing, and handling duplicate pull requests more effectively and efficiently.

To achieve our goals, we address the following research questions.

RQ1: How much effort do duplicate pull requests consume, and to what extent do they delay the review process?

Motivation: Duplicate pull requests submitted by multiple different developers are usually evaluated through the same rigorous review process as original ones. As a result, duplicate pull requests waste resources spent on separate and redundant programming and evaluation efforts. We attempt to quantify the redundant effort spent on duplicate pull requests.

RQ2: What is the context in which duplicate pull requests occur?

Motivation: As reported in prior studies [40], [98], [115], the pull-based development model is associated with higher contribution effectiveness than traditional patch-based model in terms of activity transparency and information centralization. Nevertheless, contributors are still at risk of conducting redundant development. Hence, we aim to reveal the practical factors resulting in duplicate pull requests.

RQ3: Which duplicate pull requests are more likely than their counterparts to be accepted?

Motivation: Prior research [42], [94] has studied the factors that should be examined when integrators decide whether to accept an individual pull request. However, little is known about integrators' preference for what kind of duplicate pull requests should be accepted. We hope to determine the characteristics of accepted duplicates compared to those of their counterparts and summarize the common practices of duplicate selection for integrators.

3 DATASET

In this study, we leverage our previous dataset *DupPR* [106], which contains the duplicate relations among pull requests and the profiles and review comments of pull requests from 26 popular OSS projects hosted on GitHub. We also extend the dataset by adding complementary data, including code commits, check statuses of DevOps tools and contribution histories of developers.

3.1 *DupPR* basic dataset

In our prior work [106], we have built a unique dataset of more than 2,000 pairs of duplicate pull requests (called *DupPR* [7]) by analyzing the review comments from 26 popular OSS projects hosted on GitHub. Each pair of duplicates in *DupPR* is represented in a quaternion as $\langle \text{proj}, \text{pr1}, \text{pr2}, \text{idn_cmt} \rangle$ (pr1 was submitted before pr2). Item *proj* indicates the project (e.g., *rails/rails*) that the duplicate pull requests belong to. Items *pr1* and *pr2* are the tracking numbers of the two pull requests, respectively. Item *idn_cmt* is a review comment of either *pr1* or *pr2*, which is used by reviewers to state the duplicate relation between *pr1* and *pr2*. The dataset meant to only contain the accidental duplicates of which all the authors were not aware of the other similar pull requests when creating their own. In order to increase the accuracy of this study, we recheck the dataset again and filter out the intentional duplicates that were not found before. Specifically, we omit duplicates from the dataset when they fit one of the following criteria: *i*) The authors' discussion on the associated issue reveals that the duplication was on purpose. A representative comment indicating intentional duplication is "I saw your PR and there wasn't any activity or follow up in that from last 18 days, [so I create a new one.]"¹; *ii*) The submitter of *pr2* has performed actions on *pr1*, including commenting, assigning reviewers and adding labels, which clearly indicates that s/he was aware of *pr1* before submitting her/his own one; and *iii*) The author of *pr2* immediately (< 1 min) mentioned *pr1* after creating *pr2*, which means that the author might

1. The sources of pull requests, issues and comments cited in this paper can be found online at <https://github.com/whystar/DupPR-cited>

already know that pull request before. Overall, we eliminate 330 pairs from *DupPR*.

A pull request might be duplicate of more than one pull request. Therefore, in this study, we organize a group of duplicate pull requests in a tuple structure $\langle dup_1, dup_2, dup_3, \dots, dup_n \rangle$ in which the items are sorted by their submission time. In total, we have 1,751 tuples of duplicate pull requests. Table 1 presents the quantitative overview of the dataset. It lists the main metrics including the number of pull requests, the number of pull request contributors, the number of pull request reviewers and their review comments, and the number of pull request checks (introduced in Section 3.2.2).

TABLE 1: The quantitative overview of the dataset

Metrics	Overall pull requests	<i>DupPR</i>
#Pull requests	333,200	3,619
#Contributors	39,776	2,589
#Reviewers	24,071	2,830
#Comments	2,191,836	39,945
#Checks	364,646	4,413

3.2 Collecting complementary data

3.2.1 Patch detail

GitHub API (`/repos/:owner/:repo/pulls/:pull_number/commits`) allows us to retrieve the commits on each pull request. From the returned results, we parse the sha of each commit and request the API (`/repos/:owner/:repo/commits/:commit_sha`) to return detailed information about a commit, including author and author_date. Moreover, the API (`/repos/:owner/:repo/pulls/:pull_number/files`) returns the files changed by a pull request, from which we can parse the filename and changes (lines of code added and deleted) of each changed file.

3.2.2 Check statuses

Various DevOps tools are seamlessly integrated and widely used in GitHub; examples are Travis-CI [14] for continuous integration and Code-Climate [4] for static analysis. When a pull request has been submitted or updated, a set of DevOps tools are automatically launched to check whether the pull request can be safely merged back to the codebase. GitHub API (`/repos/:owner/:repo/commits/:ref/status`) returns the check statuses for a specific commit. There are two different levels of statuses in the returned results. Because multiple DevOps tools can be used to check a commit, each tool is associated with a check status, which we call the *context-level check status*. For each context-level check status, we can parse the *state* and *context* fields. The state of a check can be designated *success*, *failure*, *pending*, or *error*. State *success* means a check has successfully passed, while *failure* indicates that the check has failed. If the check is still running and no result is returned, its *state* is *pending*. State *error* indicates a check did not successfully run and produced an error. Following the guidelines of prior work [22], [81], we treat the state *error* as the same as *failure*, which are both opposed to *success*. The context

indicates which tool is used in a specific check. According to the bot classification defined in prior study [102], the checking tools can be classified into three categories: CI (report continuous integration test results, e.g., Travis-ci), CLA (ensure license agreement signing e.g., cla/google), and CR (review source code e.g., coverage/coveralls and codeclimate). Based on all context-level check statuses of a commit, the API also returns a overall check status of that commit [8], which we call the *commit-level check status*. The state of a commit-level check can be one of *success*, *failure* and *pending*.

3.2.3 Timeline events

GitHub API (`/repos/:owner/:repo/issues/:issue_number/events`) returns the events triggered by activities (e.g., assigning a label and posting a comment) in issues and pull requests. We request this API for each pull request. From the returned result, we can parse who (actor) triggered which event (event) at what time (created_at). For close events, we can parse which commit (commit_id, aka SHA) closed the pull request. Events data are mainly used for rechecking dataset and determining pull request acceptance.

3.2.4 Contribution histories

Rather than requesting the GitHub API, we use the GHTorrent dataset [40], which makes it easier and more efficient to obtain the entire contribution history for a specific developer in GitHub. GHTorrent stores its data in several tables and we mainly use *pull_requests* (PR), *issues*, *pull_request_history* (PRH), *pull_request_comments* (PRC), and *issue_comments* (ISC). From table PR, table PRH, and table PRC, we can parse who (PRH.actor_id) submitted which pull request (PR.pullreq_id) to which project (PR.base_repo_id) at what time (PRH.created_at) and who (PRC.user_id) have commented on that pull request at what time (PRC.created_at). Similarly, from table issues and table ISC, we can parse who (issues.reporter_id) reported which issue (issues.issue_id) to which project (issues.repo_id) at what time (issues.created_at) and who (ISC.user_id) commented on that issue at what time (ISC.created_at). Based on this information we can acquire the whole contribution history for a specific developer.

3.2.5 Popularity and reputation

GHTorrent also provides tables relating to project popularity and developer reputation. From table *watchers*, we can parse who (user_id) started to star which project (repo_id) at what time (created_at). From table *projects*, we can parse which project (id) was forked from which project (forked_from) at what time (created_at). From table *followers*, we can parse who (followers) started to follow whom (user_id) at what time (created_at).

4 THE IMPACT OF DUPLICATE PULL REQUESTS

Although duplicate pull requests can bring certain benefits (e.g., they could complement each other and be combined to achieve a better solution), most unintentional duplicate

pull requests are likely to waste resources during the asynchronous development and review process. In this section, we report a quantitative analysis that helps to better understand the impact of duplicate pull requests on development efforts and review processes.

4.1 Redundant effort

Duplicate pull requests are organized as tuple structure in our dataset, *i.e.*, $\langle dup_1, dup_2, dup_3, \dots, dup_n \rangle$. For each tuple, we identify the first received pull request (*i.e.*, dup_1) as the ‘master’, and the following ones (*i.e.*, $dup_i, i > 1$) as its ‘duplicates’. Since we want to quantify how much extra effort would be costed if the first contribution has been qualified, we accumulate the effort spent on all duplicates (*i.e.*, $\sum_{i=2}^n dup_i$) as the redundancy. In this section, we analyze the redundant effort caused by duplicate pull requests from three perspectives, *i.e.*, code patch, code review, and DevOps checking.

Code patch redundancy: A group of pull requests being duplicate means that multiple contributors have spent unnecessary redundant effort on implementing similar functionalities. We measure contribution effort for the code patch of a pull request with the number of changed files and LOCs (*i.e.*, lines of code). The statistics is summarized in Table 2. We can see that each group of duplicates, on average, result in redundant contribution effort of changing more than 13 files (median of 2) and 502 LOCs (median of 16).

TABLE 2: The statistics of code patch redundancy

	Min	25%	Median	75%	Max	Mean
#Files	0	1	2	3	3824	13.79
LOCs	0	4	16	70	82973	502.68

Code-review redundancy: For a group of duplicate pull requests, each one is reviewed separately until the reviewers detect the duplicate relation among them. That is, there is a detection latency of duplicate pull requests, and the review activities of the duplicates during that latency period are redundant. We define detection latency as the time period from the submission time of a pull request to the creation time of the first comment revealing the duplicate relation between it and other pull requests. Figure 3 shows the distribution of detection latency. We find that almost half of the duplicates are detected after one day, and nearly 20% of them are detected after more than one week. The later the duplicate relation is detected, the more redundant review effort would be wasted.

Next, we compute the redundant review effort wasted during the detection latency, which is measured with the number of involved reviewers and the number of comments they have made. As shown in Table 3, there are, on average, more than 2 reviewers (median of 2) participating in the redundant review discussions and making more than 5 review comments (median of 3) before the duplicate relation is identified. This considerable redundancy cost reaches the standard number of contemporary peer review practices [74], [75] (*i.e.*, median of 2 reviewers and 2-5 comments). Considering that the availability of reviewers

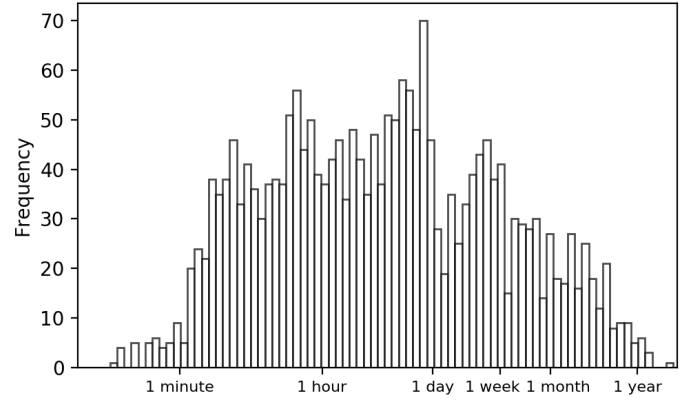


Fig. 3: Detection latency of duplicate pull requests.

has been discussed as one of the bottlenecks in large OSS projects’ review process [107], code review redundancy can be avoided through automatic detection tools [57], [73] at submission time.

TABLE 3: The statistics of code-review redundancy

	Min	25%	Median	75%	Max	Mean
#Reviewers	1	1	2	3	35	2.30
#Comments	1	2	3	6	148	5.68

DevOps redundancy: DevOps techniques, *e.g.*, continuous integration, are widely used to improve code quality in GitHub. However, running DevOps services to check pull requests consumes a certain amount of resources and time. Moreover, both newly submitted pull requests and updates on existing pull requests trigger the launch of DevOps services. Therefore, duplicate pull requests waste valuable DevOps resources on redundant checking effort. In this paper, we measure the DevOps effort on a pull request by counting the total number of commit-level checks and context-level checks on this pull request. Table 4 lists the statistics for DevOps redundancy related to duplicates. We find that each group of duplicate pull requests, on average, cause 1.34 redundant commit-level checks and 2.87 redundant context-level checks. Thus, it is possible to improve DevOps efficiency by stopping or postponing the unnecessary checks of duplicates from the scheduling queue.

TABLE 4: The statistics of DevOps redundancy

	Min	25%	Median	75%	Max	Mean
#CMT-Check	0	1	1	1	30	1.34
#CTT-Check	0	1	1	3	66	2.87

CMT-Check: commit-level checks; CTT-Check: context-level checks

4.2 Delayed review process

The review duration, the number of reviewers and the number of comments made by these reviewers are important metrics for the efficiency of the pull request review process. Based on these metrics, we study whether the

review process of duplicate pull requests differs from that of non-duplicate pull requests. We classify the duplicate pull requests into two groups: *MST* including the ‘master’ in each tuple, *DUP* including the ‘duplicates’ in each tuple. For comparison, we also create the *NON* group, which includes all non-duplicate pull requests from the corresponding projects. Figures 4, 5, and 6 plot the statistics of the review duration, the number of reviewers and the number of review comments of pull request in each group, respectively. We observe that *MST* has notably longer review duration (median: 291.07 hours), compared with *NON* (median: 22.83 hours) and *DUP* (median: 21.75 hours). Moreover, compared with *NON*, both *MST* and *DUP* have more reviewers (medians: 3 vs. 2, 3 vs. 2) and more review comments (medians: 6 vs. 3, 4 vs. 3). Furthermore, we use the \tilde{T} -procedure [50] to compare their distributions pairwise. We opt for \tilde{T} because it is robust against unequal population variances and does not have the drawbacks of two-steps methods [95]. As shown in Table 5, all distributions are statistically different (p -value < 0.05), except for *NON-DUP* of review duration. And the signs of estimators also coincide with the observations from the figures, e.g., the estimator of *NON-MST* in terms of review duration comparison is less than 0 (-0.232) which indicates that *MST* has longer review time than *NON*. This suggests that the review process of duplicate pull requests is significantly delayed and involves more reviewers for extended discussions.

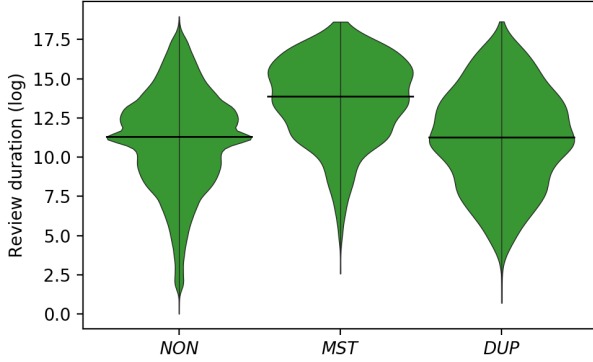


Fig. 4: The review duration of pull requests.

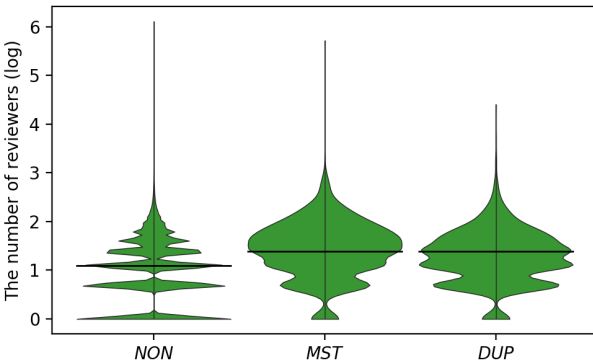


Fig. 5: The number of pull request reviewers.

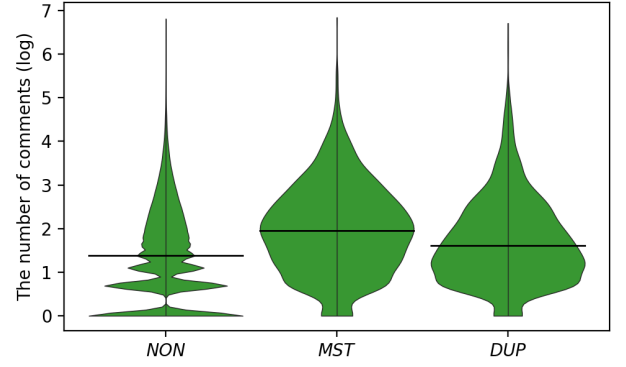


Fig. 6: The number of review comments on pull requests.

TABLE 5: Results of multiple contrast test procedure for review process measures

Pair	Estimator	Lower	Upper	Statistic	p-value
Review duration					
<i>NON - MST</i>	-0.232	-0.246	-0.218	-38.822	0.000 ***
<i>NON - DUP</i>	-0.012	-0.028	0.004	-1.771	0.172
<i>MST - DUP</i>	0.220	0.200	0.240	25.400	0.000 ***
Number of reviewers					
<i>NON - MST</i>	-0.191	-0.206	-0.176	-30.065	0.000 ***
<i>NON - DUP</i>	-0.114	-0.127	-0.100	-19.549	0.000 ***
<i>MST - DUP</i>	0.077	0.056	0.098	8.606	0.000 ***
Number of review comments					
<i>NON - MST</i>	-0.168	-0.182	-0.154	-27.077	0.000 ***
<i>NON - DUP</i>	-0.085	-0.098	-0.071	-14.338	0.000 ***
<i>MST - DUP</i>	0.083	0.063	0.104	9.305	0.000 ***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

RQ1: Duplicate pull requests result in considerable redundancy in writing code and evaluation. On average, each group of duplicate pull requests would result in code patch redundancy of more than 13 files and 500 lines of code, code-review redundancy of more than 5 review comments created by more than 2 reviewers, and DevOps redundancy of more than 1 commit-level check and more than 2 context-level checks. Moreover, duplicate pull requests significantly slow down the review process, requiring more reviewers for extended discussions.

5 CONTEXT WHERE DUPLICATE PULL REQUESTS ARE PRODUCED

Despite of the increased activity transparency and information centralization in the pull-based development, developers still submitted duplicates. Thus, we further investigate the context in which duplicates occur and the factors leading pull requests to be duplicates.

First, we investigate the context of pull requests when duplicate occurs, as described in Section 5.1. In particular, we examine the lifecycle of pull requests and discover three types of sequential relationship between two duplicate pull requests. For each relationship, we investigate whether

contributors' work patterns and their collaborating environment have any flaw that may produce duplicate pull requests.

Second, we investigate the differences between duplicate and non-duplicate pull requests, as described in Section 5.2. We identify a set of metrics from prior studies to characterize pull requests. We then conduct comparative exploration and regression analysis to examine the characteristics that can distinguish duplicate from non-duplicate pull requests.

5.1 The context of duplicate pull requests

The entire lifecycle of a pull request consists of two stages: *local creation* and *online evaluation*. In the local creation stage, contributors edit the files and commit changes to their local repositories. In the online evaluation stage, contributors submit a pull request to notify the integrators of the original repository to review the committed changes online. These two stages are separated by the submission time of a pull request. For each pair of pull requests, there are only three types of sequential relationships in logic when comparing the order in which they enter each stage. We manually analyze contributors' work patterns, discussion and the collaborating environment to explore the possible context of duplicates in different relationships. In the following sections, we first elaborate on the three types of sequential relationships and then present the identified context of duplicate pull requests demonstrated by statistics and representative cases.

5.1.1 Types of sequential relationship

We first introduce two critical time points, $T\text{-Creation}$ and $T\text{-Evaluation}$, in the lifecycle of pull requests; these time points are defined as follows.

- $T\text{-Creation}$ indicates the start of pull request local creation. It is impossible to know the exact time at which a developer begins to work since developers only `git commit` when their work is finished rather than when the work is launched. However, we can still get an approximate start time. We set $T\text{-Creation}$ as the `author_date` of the first commit packaged in a pull request, which is the earliest timestamp contained in the commit history of a pull request.
- $T\text{-Evaluation}$ indicates the start of pull request online evaluation, i.e., the submission time of a pull request. This value is the `created_at` value of a pull request.

For a pair of duplicate pull requests $\langle mst_pr, dup_pr \rangle$ (mst_pr is submitted earlier than dup_pr), we suppose that the contributor of mst_pr begins to work at $T\text{-Creation}_{mst}$ and submits mst_pr at $T\text{-Evaluation}_{mst}$, and the contributor of dup_pr starts to work at $T\text{-Creation}_{dup}$ and submits dup_pr at $T\text{-Evaluation}_{dup}$. We discover three possible sequential relationships between mst_pr and dup_pr , as shown in Figure 7.

- *Exclusive*. $T\text{-Creation}_{mst} < T\text{-Evaluation}_{mst} < T\text{-Creation}_{dup} < T\text{-Evaluation}_{dup}$, i.e., the author of dup_pr begins to work after the author of mst_pr has already finished the local work and submitted the pull request.
- *Overlapping*. $T\text{-Creation}_{mst} < T\text{-Creation}_{dup} \leq T\text{-Evaluation}_{mst} < T\text{-Evaluation}_{dup}$, i.e., the author

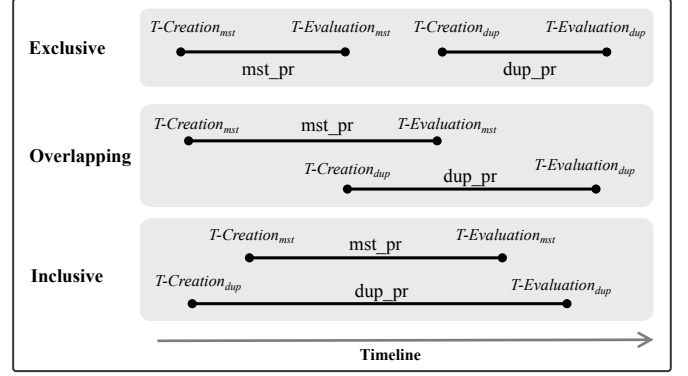


Fig. 7: The sequential relationship between two pull requests mst_pr and dup_pr .

of dup_pr starts working after the author of mst_pr starts working and before the author of mst_pr finishes working.

- *Inclusive*. $T\text{-Creation}_{dup} \leq T\text{-Creation}_{mst} < T\text{-Evaluation}_{mst} < T\text{-Evaluation}_{dup}$, i.e., although the author of dup_pr starts to work earlier than the author of mst_pr does, s/he submits the pull request later.

In the above, we discuss the common cases where developers first commit changed code and then submit a pull request. However, we also find rare cases where the pull request submission time is earlier than the first code-committing time. This might be due to the permission to submit 'empty' pull requests in the early stage of GitHub as indicated in its official document [15], or the incomplete record of developers' updates to the pull request, e.g., force-push actions. The aforementioned definition of sequential relationship between CTS (commit-then-submit) pull requests does not apply to these STC (submit-then-commit) pull requests. To demonstrate this kind of situation, we define $T\text{-Exposure}$ to indicate the exposure time of developer's ideas for STC pull requests. $T\text{-Exposure}$ is also set to be the `created_at` of pull requests. Next, we discuss the sequential relationship between two pull requests, of which at least one is a STC pull request.

TABLE 6: Three cases involving the STC pull requests

Case	mst_pr	dup_pr
1	CTS	STC
2	STC	STC
3	STC	CTS

As shown in Table 6, there are three specific cases involving STC pull requests. In case 1 ($T\text{-Creation}_{mst} < T\text{-Evaluation}_{mst} < T\text{-Exposure}_{dup}$) and case 2 ($T\text{-Exposure}_{mst} < T\text{-Exposure}_{dup}$), the local work or the idea of mst_pr has been exposed to the community before dup_pr is submitted. Therefore, we treat the sequential relationship between a pair of pull requests in these two cases as exclusive. In case 3, there are two possible situations: a) $T\text{-Exposure}_{mst} < T\text{-Creation}_{dup} < T\text{-Evaluation}_{dup}$

means that the idea of *mst_pr* has been exposed before the author of *dup_pr* starts to work and their sequential relationship can be seen as exclusive. b) $T\text{-}Creation_{dup} < T\text{-}Exposure_{mst} < T\text{-}Evaluation_{dup}$ means the author of *dup_pr* starts to work before the idea of *mst_pr* is exposed and finishes the work after that; therefore, their sequential relationship can be seen as inclusive.

Finally, to explore the distribution of the three types of relationships in our dataset, we convert each tuple of duplicate pull requests ($\langle dup_1, dup_2, dup_3, \dots, dup_n \rangle$) to pairs: (dup_i, dup_j) , where $1 \leq i, j \leq n$ and $i < j$. Table 7 shows the distribution, and we can see that the majority of duplicate pull request pairs have an exclusive sequential relationship (i.e., the duplicate contribution begins to work after the original pull request has been visible), which suggests that it is still a great challenge of awareness and transparency [37] during collaboration process.

TABLE 7: The distribution of different sequential relationships in the dataset

	Exclusive	Overlapping	Inclusive
Count	1,924	17	81

5.1.2 Context of exclusive duplicate pull requests

Not searching for existing work. For a pair of exclusive duplicate pull requests, there is a time window during which the author of *dup_pr* had a chance to figure out the existence of *mst_pr*. However, the author failed to do so and finally submitted a duplicate pull request. For example, contributors did not search the existing pull requests for similar work (e.g., the typical responses in duplicates: “Oh, Sorry I did not search for a previous PR before submitting a PR” and “Ah should have searched first, thanks”). In some cases, developers’ search was not complete because they only searched the open pull requests and missed the closed ones (e.g., “Ah, my bad. I thought I searched, but I must have only been looking at open”). The survey conducted by Gousios [41] also showed that 45% contributors occasionally or never check whether similar pull requests already exist before coding.

Diversity of natural language usages. Some developers tried to search for existing duplicates, but they ultimately found nothing (e.g., “Sorry, I searched before pushing but did not find your PR...”). One challenge is the diversity of natural language usages. For a pair of duplicate pull requests, we compute the common words ratio based on their titles, which is calculated by the following formula.

$$CWR(mst_pr, dup_pr) = \frac{|WS_{mst_pr} \cap WS_{dup_pr}|}{|WS_{mst_pr}|} \quad (1)$$

WS_{mst_pr} and WS_{dup_pr} represent the set of words extracted from the titles of *mst_pr* and *dup_pr*, respectively, after necessary preprocessing like tokenizing, stemming [61], and removing common stop words. Figure 8 shows the statistics of common words ratio. Approximately half of them have a value less than 0.25, which means a pair of duplicates tend to share a small proportion of common

words. That is to say, a keyword-based query cannot always successfully detect existing duplicate pull requests due to the difference in wording for the same concept. For example, the title of *angular/angular.js/#4916* is “Fixed step12 correct file reference” and the title of *angular/angular.js/#4860* is “Changed from phone-list.html to index.html”. We can see that the two titles share no common word although the two pull requests have edited the same file and changed the code in the same line.

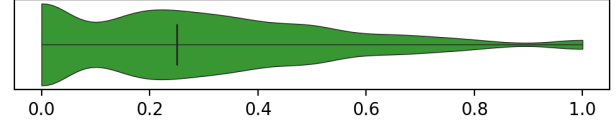


Fig. 8: The statistics of common words ratio between duplicates.

Disappointing search functionality in GitHub. Another challenge that can cause ineffective searching for duplicates is that GitHub’s search functionality might be disappointing in retrieving similar pull requests even though they share common words. For example, the titles of *angular/angular.js/#5063* and *angular/angular.js/#7846* are “fix(copy): preserve prototype chain when copying object” and “Use source object prototype in object copy”, respectively, which share three common critical words, i.e., *prototype*, *copy*, and *object*. For testing purpose, we launch a query in GitHub using the keywords *prototype copy object*. We retrieve 9 pages (each page containing 10 items) of issues and pull requests in the search results and we finally find *angular/angular.js/#5063* in the 7th page. It is unlikely that developers have the willingness and patience to look through 7 pages of search results to figure out the existence of duplicates, since people tend to focus on the first few pages [49]. Perhaps that is exactly what leads the author of *angular/angular.js/#7846* to submit a duplicate, although he blamed the failed retrieval on himself (“apologies, I did search before posting (forget the search term I used) but clearly my search was bad... Thanks for finding the dup”).

Large searching space. Developers might manually look through the issue tracker to search for duplicates rather than retrieving through a query interface. Sometimes it is hard to find out the existing duplicates due to large searching space. The statistics of exclusive intervals between duplicates is listed in Table 8. On average, the local work of *dup_pr* is started approximately 1,400 hours (i.e., more than 58 days) after *mst_pr* has been submitted. During that long period, many new pull requests have been submitted in popular projects. For example, 307 pull requests were submitted between *pandas-dev/pandas/#9350* and *pandas-dev/pandas/#10074*. These pull requests can occupy more than 10 pages in the issue tracker, which makes it rather hard and ineffective to review historical pull requests page by page, as a developer stated “...This is a dup of that PR. I should have looked harder as I didn’t see that one when I created this one...”.

Overlooking linked pull requests. When developers submit a pull request to solve an exiting GitHub issue, they can build a link between the pull request and the issue by

TABLE 8: The statistics of exclusive intervals (in hour)

	Min	25%	Median	75%	Maxs	Mean
Interval	0.004	23.81	212.59	1276.82	29377.12	1397.59

referencing the issue in the pull request description. The cross-reference is also displayed in the discussion timeline of the issue. Links not only allow pull request reviewers to find out the issue to be solved by a pull request but also help developers who are concerned with an issue to discover which pull requests have been submitted for that issue. In some cases, contributors did not examine or did not notice the linked pull requests to an issue (e.g., “Uhm yeah, didn’t spot the reference in#21967” and “Argh, didn’t see it in the original issue. Need more coffee I guess”) to make sure that no work had already been submitted for that issue, and consequently submitted a duplicate pull request.

Lack of links. If a developer does not link her/his pull request to the associated issue, other developers might asynchronously do the duplicate work to fix the same one. For example, a developer *Dev2* submitted a pull request *facebook/react/#6135* trying to address the issue *facebook/react/#6114*. However, *Dev2* was told that a duplicate pull request *facebook/react/#6121* was already submitted by *Dev1* before him. The conversation between *Dev1* and *Dev2* (*Dev2* said “I’m glad to hear that. But please link your future PRs to the issues”, and *Dev1* replied “Yeah I will, that’s on me!”) revealed that the lack of the link accounted for the duplication.

Missing notifications. If developers have watched [37], [80] a project, they receive notifications about events that occur in the project, e.g., new commits and pull requests. The notifications are displayed in developers’ GitHub dashboard and, if configured, sent to developers via email. However, developers might miss the important notifications due to information overload [36], and eventually submit a duplicate. For example, *kubernetes/kubernetes/#43902* was duplicate of *kubernetes/kubernetes/#43871* because the author of *kubernetes/kubernetes/#43902* missed the creation notification of the early one (as the author said “missed the mail for the PR it seems :-/”).

5.1.3 Context of overlapping and inclusive duplicates

Unawareness of parallel work. Developers who encounter a problem might prefer to fix the problem by themselves and submit a pull request, instead of reporting the problem in the issue tracker and waiting for a fix. When a problem is encountered by two developers at the same time, regardless of which developer is the first to work on the problem, the other developer might also start to work on the problem before the first developer submits a pull request. In such cases, both developers are unaware of concurrent activities of each other, because their local work is conducted offline and is not publicly visible. For example, the authors of *emberjs/ember.js/#4214* and *emberjs/ember.js/#4223* individually fixed the same typos in parallel without being aware of each other, and finally submitted two duplicate pull requests.

Implementing without claiming first. Sometimes, developers directly start to implement a patch for a GitHub issue without claiming (e.g., leaving a comment on the corresponding issue like “I’m on it”). This can introduce a risk that other interested developers might also start to work on the same issue without awareness of that there is already a developer working on that issue. For example, although two developers were both trying to solve the issue *facebook/react/#3948*, neither of them claimed the issue before coding their patch. Finally, they submitted two duplicate pull requests *facebook/react/#3949* and *facebook/react/#3950*. The phenomenon that developers are not used to claim issues was also reported in previous research [114].

Missing existing claims. Although a public issue has been claimed by a developer, other OSS contributors still have a chance of missing the claim comments among issue discussions. For example, a developer *Dev1* first claimed the issue *scikit-learn/scikit-learn/#8503* by leaving a comment “We are working on this”. However, another developer *Dev2* did not notice this claim as explained by herself: “Ah, nope. I just realized someone was also working on it after I committed”. Consequently, *Dev2* and *Dev1* conducted duplicate development in parallel and submitted two duplicate pull requests *scikit-learn/scikit-learn/#8517* and *scikit-learn/scikit-learn/#8518*, respectively.

Overlong local work. For overlapping and inclusive duplicate pull request pairs, we calculate the local duration of the work started earlier. Specifically, we collect two groups of pull requests: (a) *OVL*, which includes *mst_pr* of each pair of overlapping duplicates, and (b) *INC*, which includes *dup_pr* of each pair of inclusive duplicates. Figure 9 plots the duration statistics of each group together along the group *NON*, which includes all non-duplicate pull requests. We observe that compared with the pull requests in *NON*, the pull requests in *OVL* and *INC* have longer local durations regarding median measures. We also test the difference using the \bar{T} -procedure test. As shown in Table 9, the difference is significant ($p\text{-value} < 0.05$), and the signs of estimators present consistent difference directions. This reveals that overlong local work delays the exposure time of work and thereby hinders late contributors from realizing in a timely fashion that someone has already done the same work. As discussed in [41], developers rarely recheck the existence of similar pull request after they have finished the local work.

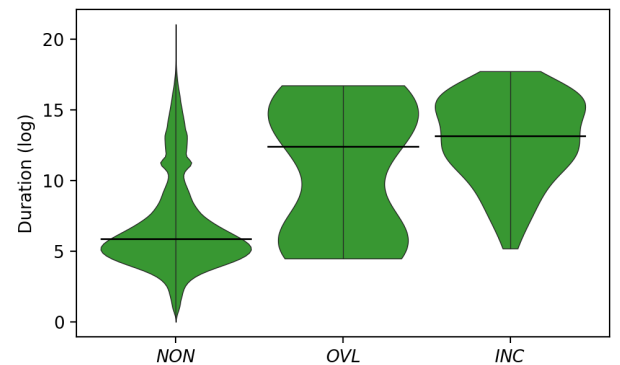


Fig. 9: Duration of local work in each group.

TABLE 9: Results of multiple contrast test procedure for local work durations

Pair	Estimator	Lower	Upper	Statistic	p-value
OVL - NON	0.256	0.109	0.402	4.188	0.001 ***
NON - INC	-0.385	-0.427	-0.344	-22.251	0.000 ***
OVL - INC	-0.130	-0.294	0.035	-1.891	0.138

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

RQ2-1: We identified 11 contexts where duplicate pull requests occur, which are mainly relating to developers' behaviors, e.g., not checking for existing work, not claiming before coding, not providing links, and overlong local work, and their collaborating environment, e.g., unawareness of parallel work, missing notifications, lack of effective tools for checking for duplicates.

5.2 The difference between duplicate and non-duplicate pull requests

Although some specific cases could be effectively avoided if developers pay attention to their work patterns, duplicates are difficult to eradicate completely considering the distributed and spontaneous nature of OSS development. Therefore, automatic detection of duplicates is still needed to help reviewers dispose of duplicates faster and in a timely fashion. Given that prior studies have mainly used a similarity-based method to detect duplicate pull requests [57], [73], we are interested in exploring the difference between duplicate and non-duplicate pull requests from a comparative perspective, which could offer useful guidance to optimize detection performance. In particular, we want to observe what distinguishing characteristics of duplicate pull requests are leading them to be duplicates. First, we identify metrics that are used in prior research, as shown in Section 5.2.1. Then, we compare duplicate and non-duplicate pull requests in terms of each metric, and check whether significant difference could be observed between them through statistical test, as described in Section 5.2.2. Furthermore, in Section 5.2.3, we apply a regression analysis to model the correlations between the collected metrics and pull requests' likelihood of being duplicates.

5.2.1 Metrics

As the difference between duplicates and non-duplicates has not been studied in past studies, our study is an exploratory analysis. Therefore, we identified highly related metrics which have been studied in the previous research in the area of OSS contribution including traditional patch-based development [20], [44], [47], [68], [101] and modern pull request development [27], [40], [52], [71], [93], [95], [96], [108], [114]. The selected metrics are classified into the following three categories.

Project-level characteristics.

Maturity. Previous studies [71], [93], [108] used the metric `proj_age`, i.e., the period of time from the time the project was hosted on GitHub to the pull request submission time, as an indicator of the project maturity.

Workload. Prior studies have characterized project workload using two metrics: `open_tasks` [108] and `team_size` [40], [93], [108], which are the number of open issues and open pull requests at the pull request submission time and the number of active core team members during the last three months, respectively.

Popularity. In measuring project popularity, the metrics `stars` and `forks`, i.e., the total number of stars and the total number of forks the project has got prior to the pull request submission, were commonly used in previous studies [27], [93].

Hotness. This metric is the number of total changes on files touched by the pull request three months before the pull request creation time [40], [108].

Submitter-level characteristics.

Experience. Developers' experience before they submit the pull request has been analyzed in prior studies [40], [47]. This measure can be computed from two perspectives: project-level experience and community-level experience. The former measures the number of previous pull requests that the developer have submitted to a specific project (`prev_pullreqs_proj`) and their acceptance rate (`prev_prs_acc_proj`). The latter measures the number of previous pull requests that the developer have submitted to GitHub (`prev_pullreqs`) and their acceptance rate (`prev_prs_acc`). When calculating acceptance rate, the determination of whether the pull request was integrated through other mechanisms than GitHub's merge button follows the heuristics defined in previous studies [40], [114]. We also use two metrics `first_pr_proj` and `first_pr` to represent whether the pull request is the first one submitted by the developer to a specific project and GitHub, respectively.

Standing. A dichotomous metric `core_team`, which indicates whether the pull request submitter is the core team member of the project, was commonly used as a signal of the developer's standing within the project [93], [108]. Furthermore, a continuous metric `followers`, i.e., the number of GitHub users that are following the pull request submitter, was used to represent the developers' standing in the community [40], [93], [108].

Social connection. The metric `prior_interaction`, which is the total number of events (e.g., such as commenting on issues and pull requests) prior to the pull request submission that the developer has participated in within the project, was usually used to measure the social connection between the developer and the project [93], [108].

Patch-level characteristics.

Patch size. Prior studies [40], [93], [95] quantified the size of a patch, i.e., the changes contained in the pull request, in different granularity. The commonly used metrics are the number of changed files (`files_changed`) and the number of changed lines of code added and deleted (`loc`).

Textual length. This metric is computed by counting the number of characters in the pull request title and description [108].

Issue tag. This metric indicates whether the pull request description contains links to other GitHub issues or pull requests [40], [108], such as "fix issue #1011". We determine

this metric by automatically checking the presence of cross-references in the pull request description based on regular expression technique.

Type. Prior studies [44], [63] summarized that developers can make three primary types of changes: fault repairing (FR), feature introduction (FI), and general maintenance (GM). The change type (*change_type*) of the pull request is identified by analyzing its title and commit messages based on a set of manually verified keywords [63]. Prior studies [45], [97] also identified the types of developer activities on the basis of the types of changed files. We follow the classification by Hindle *et al.* [45], which includes four types: changing source code files (*Code*), changing test files (*Test*), changing build files (*Build*), and changing documentation files (*Doc*). This metric (*activity_type*) is determined by checking the names and extensions of the files changed by the pull request.

5.2.2 Comparative exploration

In order to explore the difference between duplicate and non-duplicate pull requests, we compare them in terms of each of the collected metrics and study to what extent a metric varies across duplicate and non-duplicate pull requests. Specifically, we formulate a non-directional hypothesis which can be used when there is insufficient theory basis for the exact prediction (*i.e.*, we do not predict the exact direction of the difference). The null hypothesis and the alternative hypothesis are defined as follows:

H_0 : duplicate and non-duplicate pull requests exhibit the same value of metric m .

H_1 : duplicate and non-duplicate pull requests exhibit different values of metric m .

$\forall m \in \{\text{proj_age, open_tasks, team_size, forks, stars, hotness, prev_pullreqs, prev_prs_acc, first_pr, first_pr_proj, prev_pullreqs_proj, prev_prs_acc_proj, core_team, followers, prior_interaction, loc, files_changed, text_len, issue_tag, change_type, activity_type}\}$

H_0 is tested with *Mann-Whitney-Wilcoxon* test [105] on continuous metrics and *Chi-square* test [72] on categorical metrics. The test results are listed in Table 10 which reports the p-value and effect size of each test. The p-values are adjusted using the *Benjamini-Hochberg (BH)* method [23] to control the false discovery rate. To measure the effect size, we use *Cliff's delta (d)* [59] as it is a non-parametric approach which does not require the normality assumption of a distribution.

We reject H_0 and accept H_1 when p-value is less than 0.05. We can see that the null hypothesis is rejected on all metrics except for *open_tasks*. This means that duplicate and non-duplicate pull requests are significantly different in terms of all metrics except for *open_tasks*. Following the previous guidelines [70], [90] on interpreting the effect size (trivial: $|d| \leq 0.147$; small: $0.147 < |d| < 0.33$; medium: $0.33 \leq |d| < 0.474$; large: $|d| \geq 0.474$), we find that the effect size of difference is generally small with a maximum of 0.285.

5.2.3 Regression analysis

The comparative exploration does not consider the correlations between metrics. As a refinement, we apply a

TABLE 10: Results of hypothesis test

Metric	Adjusted p-value	Effect size
Project-level characteristics		
proj_age	4.1e-21 ***	0.091
open_tasks	0.791	0.003
team_size	3.8e-41 ***	0.131
stars	2.1e-46 ***	0.139
forks	8e-61 ***	0.159
hotness	4.5e-36 ***	0.122
Submitter-level characteristics		
first_pr	9.9e-33 ***	0.045
prev_pullreqs	7.7e-94 ***	0.199
prev_prs_acc	8.5e-90 ***	0.205
first_pr_proj	6.4e-148 ***	0.148
prev_pullreqs_proj	1.1e-190 ***	0.285
prev_prs_acc_proj	8.5e-52 ***	0.173
core_team	2.5e-116 ***	0.192
followers	1.2e-20 ***	0.090
prior_interaction	5.1e-107 ***	0.212
Patch-level characteristics		
files_changed	4.6e-08 ***	0.050
loc	3e-16 ***	0.079
text_len	9.1e-66 ***	0.166
issue_tag	0.001 **	0.027
change_type	8.5e-26 ***	0.095
activity_type	3.4e-07 ***	0.003

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

regression analysis to model the effect of the selected metrics on pull requests' likelihood of being duplicates.

Regression modeling. We build a mixed effect logistic regression model which is fit to capture the relationship between the explanatory variables, *i.e.*, the metrics discussed in Section 5.2.1, and a response variable, *i.e.*, *is_dup*, which indicates whether a pull request is a duplicate. Since our dataset is nested in the aspect of project (*i.e.*, the pull requests collected from 26 different projects), the selected metrics are modeled as fixed effects, and a new variable *proj_id* is modeled as a random effect, to mitigate the over-represented phenomena present in some of the projects. Instead of building one model with all metrics at once, we add one level metrics at a time and build a model, which can check whether the addition of the new metrics can significantly improve the model. As a result, we compare the fit of three models: a) *Model 1*, which includes only project-level variables, b) *Model 2*, which adds the submitter-level variables, and c) *Model 3*, which adds patch-level variables. In the models, all numeric factors are log transformed (plus 0.5 if necessary) to stabilize variance and reduce heteroscedasticity [60]. We manually check the distributions of all variables, and conservatively remove not more than 3% of values as outliers with exponential distributions. This reduces slightly the size of our dataset onto which we build the regression models, but ensures that our models are robust against outliers [67]. In addition, we check for the correlation of coefficients and the Variance Inflation

Factors (VIF below 5 as recommended [32]) among variables to overcome the effect of multicollinearity. Specifically, four metrics (`forks`, `prev_prs_proj`, `prev_acc_proj`, and `first_pr`) are removed due to multicollinearity. This process leaves us with 17 features, which can be seen in Table 11.

Analysis results. The analysis results are shown in Table 11. In addition to the coefficient, standard error, and significance level for each variable, the table reports the area under the ROC curve (AUC), the marginal R-squared (R_m^2) and conditional R-squared (R_c^2) to quantify the goodness-of-fit of each model. We can see that the models can explain more variability in the data when considering both the fixed and random effects ($R_c^2 > R_m^2$). Overall, Model 3 performs better than the other two Models (AUC: 0.729 vs 0.700/0.719) and they have obtained consistent variable effects (*i.e.*, there is no significant effect flipping from positive to negative and vice versa), therefore we discuss their effects based on Model 3.

With regard to project-level predictors, `open_tasks` and `team_size` have significant, positive effects. This suggests that the more open tasks (pull requests and issues) and active core team members at the submission time of a new coming pull request, the more likely the new pull request is a duplicate. Especially, `open_tasks` does not show a significant difference by making the comparison with a single hypothesis testing, but exhibits a strong positive effect when controlled for other confounds. The predictor `stars` has a strong, negative effect, which means that the more popular the project becomes the less likely the submitted pull request is a duplicate. Our explanation is that the popular projects have well-established codebase, contribution guidelines and collaborating process. Hot files tend to attract more contributions from the community, but surprisingly, there is a negative effect of the `hotness` metric. We assume that pull requests changing hot files are more likely to be reviewed faster and get accepted in a timely fashion. A quick review can allow the target issue to be solved in short time, which prevents others from encountering the same issue and submitting duplicate pull requests. We leave a deep investigation in the future work.

As for submitter-level predictors, `prev_prs_acc` has a negative effect and `first_pr_proj` has a positive effect when its value is `TRUE`. This suggests that pull requests submitted by inexperienced developers and newcomers are more likely to be duplicates. On the contrary, the predictors `prior_interaction` and `core_team` (`TRUE`) have significant, negative effects, which indicates that pull requests from core team members and developers who have a stronger social connection to the project are less likely to be duplicates. This highlights that developers equipped with enough experience and those having a stronger relationship with the project do better in avoiding duplicated work.

For patch-level predictors, two size related predictors present opposite effects. The predictor `loc` has a negative effect, which indicates that pull requests changing more lines of codes have a less chance of being duplicates. While the predictor `files_changed` has a positive effect, suggesting that pull requests changing more files are more likely to be duplicates. Generally speaking, the more lines of code a patch have changed, the more complicated and

difficult a task it solves, which poses a barrier for potential contributors (*i.e.*, decreasing the likelihood of duplication). While holding other variables constant, if a pull request has touched more files, it increases the probability of the patch being duplicate (or partial conflict) with others' code changes. We think this interesting result deserves a future investigation. The predictor `text_len` has a positive effect, indicating that pull requests with complex description are more likely to be duplicates. Longer description may indicate higher complexity and thus longer evaluation [108], which increases the likelihood of the same issue being encountered by more developers who might also submit a patch for the issue. The predictor `issue_tag` has a positive effect when its value is `TRUE`, suggesting that pull requests solving already tracked issues have greater chances of being duplicates. One possible reason is that tracked issues are already publicly visible, and they are more likely to attract more interested developers and result in conflicts. In terms of change types (`change_type`), we can see that compared with pull requests of the type `FR`, pull requests of the type `FI`, `GM` and `Other` are less likely to be duplicates. We speculate that fixing bugs are more likely to produce duplicates because bugs tend to have general effect on a bigger developer base compared to new feature or maintenance requirements which might be specific to a certain group of developers. For activity types (`activity_type`), we notice that pull requests changing test files (`activity Test`) and documentation files (`activity Doc`) have less chances of being duplicates, compared with those changing source code files. OSS projects usually encourage newcomers to try their first contribution by writing documentation and test cases [6], [11]. We conjecture that activities changing source code files might require more effort and time to conduct the local work, which are more risky and prone to duplication.

***RQ2-2:** Duplicate pull requests are significantly different from non-duplicate pull requests in terms of project-level characteristics (e.g., changing cold files and submitted when the project has more active core team members), submitter-level characteristics (e.g., submitted from newcomers and developers who have weaker connection to the project), and patch-level characteristics (e.g., solving already tracked issues rather than non-tracked issues and fixing bugs rather than adding new features or refactoring).*

6 INTEGRATORS' PREFERENCE BETWEEN DUPLICATE PULL REQUESTS.

To investigate what kind of duplicate pull requests are more likely to be accepted than their counterparts, we first construct a dataset of the integrators' choice between duplicates, as described in Section 6.1. Then we perform two investigations.

First, as described in Section 6.2, we identify metrics from prior work in the area of patch evaluation and acceptance, and apply them in a regression model to analyze their effects on the response variable *accept*, which indicates whether a duplicate pull request has been accepted or rejected.

TABLE 11: Statistical models for the likelihood of duplicate pull requests

	Model 1			Model 2			Model 3		
	response: <i>is_dup</i> = 1			response: <i>is_dup</i> = 1			response: <i>is_dup</i> = 1		
	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.
<code>log(proj_age)</code>	0.005	0.064		0.127	0.065	*	0.068	0.061	
<code>log(open_tasks + 0.5)</code>	0.247	0.043	***	0.199	0.042	***	0.192	0.042	***
<code>log(team_size + 0.5)</code>	0.152	0.080	.	0.209	0.079	**	0.256	0.080	**
<code>log(stars + 0.5)</code>	-0.046	0.013	***	-0.044	0.013	***	-0.049	0.013	***
<code>log(hotness + 0.5)</code>	-0.049	0.013	***	-0.010	0.013		-0.049	0.015	***
<code>log(prev_pullreqs + 0.5)</code>	-	-	-	-0.034	0.014	*	-0.020	0.014	
<code>log(prev_prs_acc + 0.5)</code>	-	-	-	-0.207	0.064	**	-0.221	0.064	***
<code>first_pr_proj TRUE</code>	-	-	-	0.254	0.055	***	0.231	0.055	***
<code>log(followers + 0.5)</code>	-	-	-	0.005	0.013		0.002	0.013	
<code>core_team TRUE</code>	-	-	-	-0.222	0.056	***	-0.207	0.056	***
<code>log(prior_interaction + 0.5)</code>	-	-	-	-0.035	0.011	**	-0.044	0.011	***
<code>log(files_changed + 0.5)</code>	-	-	-	-	-	-	0.098	0.030	**
<code>log(loc + 0.5)</code>	-	-	-	-	-	-	-0.049	0.015	***
<code>log(text_len + 0.5)</code>	-	-	-	-	-	-	0.120	0.017	***
<code>issue_tag TRUE</code>	-	-	-	-	-	-	0.102	0.038	**
<code>change_type FI</code>	-	-	-	-	-	-	-0.308	0.043	***
<code>change_type GM</code>	-	-	-	-	-	-	-0.368	0.060	***
<code>change_type Other</code>	-	-	-	-	-	-	-0.291	0.048	***
<code>activity_type Test</code>	-	-	-	-	-	-	-0.285	0.072	***
<code>activity_type Build</code>	-	-	-	-	-	-	0.020	0.084	
<code>activity_type Doc</code>	-	-	-	-	-	-	-0.317	0.059	***
<code>activity_type Other</code>	-	-	-	-	-	-	-0.006	0.060	
Akaike’s Information Criterion (AIC):	37451.49			37059.59			36861.77		
Bayesian’s Information Criteria (BIC):	37526.33			37198.58			37118.37		
Area Under the ROC Curve (AUC):	0.700			0.719			0.729		
Marginal R-squared (R_m^2):	0.03			0.05			0.08		
Conditional R-squared (R_c^2):	0.18			0.20			0.25		

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

Second, we want to learn what exactly integrators examine when they accept a duplicate pull request rather than its counterparts. We thus manually inspect 150 randomly selected duplicate pairs and use the card sorting method [82] to analyze the integrators’ explanations of their choice between duplicates, as described in Section 6.3.

6.1 The dataset of integrators’ choice between duplicate pull requests

We convert each duplicate pull request tuple into pairs (i.e., $\langle mst_pr, dup_pr \rangle$, mst_pr is submitted earlier than dup_pr), and collect integrators’ choice on each pair of duplicate pull requests. However, GitHub does not explicitly label which duplicate has been accepted by the integrators. Therefore, we decide to determine integrators’ choice based on pull request status. A pull request in GitHub may occupy a variety of status, i.e., *open*, *merged*, and *closed*. The status *open* means a pull request is still under review and the integrators have not made the final decision, while the

status *closed* means that the review of the pull request has concluded and should no longer be discussed. If a pull request is in the status *merged*, the pull request has been accepted and the review is over.

The dataset construction consists of two steps: (a) filtering out non-compared duplicate pairs, and (b) comparing the statuses of duplicates, as elaborated in the following sections.

6.1.1 Filtering out non-compared duplicate pairs

In the study, we focus on the duplicate pairs that integrators compared after detecting their duplicate relation. First, we exclude duplicate pairs in which both pull requests remain open. Then, we exclude duplicate pairs which were closed but not compared by integrators. Figure 10 shows the different cases where the relation identification and decision-making between duplicates happened at different times. In case A, mst_pr was closed before dup_pr was submitted. In case B, although mst_pr was closed after dup_pr

was submitted, *mst_pr* (or *dup_pr*) was closed before the duplicate relation between them was identified. In case C, the relation identification and decision-making between *mst_pr* and *dup_pr* happened after *mst_pr* and *dup_pr* were submitted and before they were closed. Duplicate pairs of cases A and B are excluded because integrators did not make a comparison between them before making decisions. Moreover, we also exclude from duplicate pairs of case C those in which one pull request was closed by its submitter before integrators left any comment. Finally, we exclude 875 non-compared duplicate pairs, and 1,147 duplicate pairs remain.

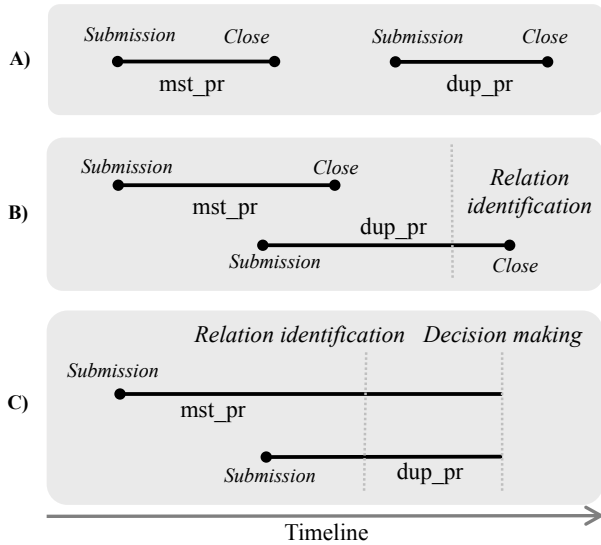


Fig. 10: Different cases where the relation identification and decision-making between duplicates happen at different times.

6.1.2 Comparing the statuses of duplicates

For a pair of duplicate pull requests, there are several combination of their statuses (e.g., one is merged and the other one is closed). In the following, we describe how to determine integrators' choice in different situations.

Only one is merged: If one of the duplicate pull requests is merged and the other one is closed or open, the merged one has clearly been accepted by integrators.

Both are merged: It is possible that both duplicates are accepted with some necessary coordination. For example, *rust-lang/rust/#20380* was first merged, and *rust-lang/rust/#20437* was then rebased and merged afterwards to provide an enhancement to the previous solution. In other cases, project integrators might also merge two duplicate pull requests to different branches; for example, *rails/rails/#28068* and *rails/rails/#28399* were merged to branches *master* and *5-0-stable*, respectively.

Both are not merged: It is somewhat complicated to determine which one has been accepted in a pair of duplicates in which neither is merged (i.e., *(closed, open)* or *(closed, closed)* or *(open, closed)*). Unlike the status *merged*, which always means that a pull request has been accepted, the status *closed* can indicate that a pull request is accepted or rejected,

which depends on the merge strategy of a specific project. In GitHub, project integrators can click the merge button on the web page to accept pull requests. However, integrators can also merge pull requests outside GitHub via several git commands. After that, integrators close the original pull requests. This means that closed pull requests might have been accepted. Prior research [40], [114] used a set of heuristics to automatically determine whether a closed pull request has been accepted. However, the heuristics are not completely reliable as some accepted pull requests might be mistakenly recognized as rejected, or vice versa. To avoid this bias, we manually examine the entire review history of a pair of duplicates to determine which one has been accepted. Additionally, it is possible that both duplicates in a pair are rejected. For example, *rails/rails/#10737* and *rails/rails/#10738* were both rejected by integrators because integrators think the change is not necessary.

Finally, as shown in Table 12, we collect a total of 1,082 duplicate pairs in which only one is accepted. Subsequently, we conduct regression analysis and manual inspection based on those 1,082 duplicate pairs.

TABLE 12: The statistics of integrators' choice between duplicates

	Accept_One	Accept_Both	Reject_Both
Count	1,082	36	29

6.2 Regression analysis

We conduct a regression analysis to investigate what factors would affect the chance that duplicate pull requests would be accepted by integrators. In the following sections, we present the selected predictors, regression models and analysis results.

6.2.1 Predictors

The predictor selection is based on prior work in the area of patch acceptance analysis. The selected predictors are split into three categories: submitter-level, patch-level, and review-level metrics. Compared with the study in Section 5.2, this study additionally includes review-level metrics because many signals in the code review process, e.g., review discussion length, are available after pull request submission, and past studies have found that review-level metrics have significant effects on pull request acceptance [40], [47], [52], [108]. However, this section does not include project-level metrics because two duplicate pull requests have the same project environment at the decision-making time. To start with, we discuss our metrics as follows.

Submitter-level metrics.

Developer experience is an important factor affecting patch acceptance. Prior studies [20], [40], [52] have shown that more experienced developers are more likely to get their patches accepted. To understand whether developer experience affects integrators' choice between duplicates, we still use six metrics to operationalize developer experience, i.e., *prev_pullreqs_porj*, *prev_prs_acc_porj*,

`prev_pullreqs`, `prev_prs_acc`, `first_pr_proj` and `first_pr`, as discussed in Section 5.2. The previous studies showed that pull requests submitted by developers with higher standing are more likely to be accepted [93], [108]. To investigate the influence of developers’ standing on integrators’ choice, we include two metrics `core_team` and `followers` defined in Section 5.2. A developer’s social relationship and interaction history with others can affect others’ judgement for the developer’s work [28], [42], [68]. Prior studies have found that pull requests from developers with a stronger social connection to the project have a higher chance to be accepted [93], [108]. In addition to the metric `prev_interaction` as defined earlier, we include the metric `social_strength`, which represents the proportion of decision-making group members that have co-occurred with the submitter in at least one discussion during the last three months, to examine the effects of social metrics on integrators’ choice.

Patch-level metrics.

Prior studies [40], [93], [108] have found that large pull requests are less likely to be accepted. Integrators value small pull requests as they are easy to assess and integrate [42]. To investigate the effect of patch size on integrators’ choice between duplicates, we include the two size-related metrics presented in Section 5.2, *i.e.*, `files_changed` and `loc`. The study conducted by Yu *et al.* [108] showed that pull requests with longer description have higher chances to be rejected. It also revealed that pull requests containing links to issues have higher acceptance rates. For this, we include the metrics `text_len` and `issue_tag`, which are already defined in Section 5.2. From the prior work [37], [42], we learn that the existence of testing code is treated as a positive signal when integrators evaluate pull requests. Pull requests with test cases are more likely to be accepted [40], [93], [108]. To investigate this, we include a dichotomous metric `test_inclusion` to indicate whether the pull request has changed test files. Finally, in the context of selection between duplicate pull requests, we conjecture that the order that pull requests arrive might affect integrators’ choice. The duplicate pull request submitted early might be more likely to be accepted than the late one, because people usually consider the recent one is redundant and should be closed in favor of the old one [12], [87], [88]. To verify our conjecture, we include a dichotomous metric `early_arrival` to indicate whether the pull request is submitted earlier than its counterpart.

Review-level metrics.

The participation metrics relating to human reviewers have been shown to affect the latency and outcome of pull request review [52], [93], [108]. For example, the study by Tsay *et al.* [93] showed that pull requests with a higher amount of comments are less likely to be accepted. To investigate the effects of human participation metrics on integrators’ choice, we include two discussion-related metrics, *i.e.*, the total number of comments on the pull request (`comments`) and the number of inline comments pointing to the source code (`comments_inline`). In addition, developers might leave tendentious comments on a pull request, according to whether they like or dislike a pull request [16], [94]. Following the philosophy of so-

cial coding [21], it is interesting to analyze whether the duplicate pull requests received more positive comments would likely win out, compared to those with more negative comments. To verify the hypothesis, we include two metrics: the proportion of the comments expressing positive sentiment (`comments_pos`) and the proportion of the comments expressing negative sentiment (`comments_neg`). To analyze the sentiment in pull request comments, we use the state-of-the-art sentiment analysis tool Senti4SD [30] retrained on the latest GitHub data [66]. Integrators also rely on automated testing tools to check pull request quality [42]. For example, prior study [108] found that the presence of CI test failure has a negative effect on pull request acceptance. To investigate this, we include three metrics `CI`, `CLA`, and `CR`, to represent the check statuses of the most recent commit in the pull request, which are returned by the three kinds of check tools discussed in Section 3. Finally, we include a metric `revisions`, *i.e.*, how many times the pull request has been updated, to indicate the maturity of and the efforts that developers put on the pull request. The prior study [47] showed that patch maturity is one of the major factors affecting patch acceptance.

TABLE 13: Overview of metrics

Metric	Mean	St.	Min	Median	Max
Submitter-level characteristics					
<code>prev_pullreqs</code>	1.7e2	3.1e2	0	45.00	3.5e3
<code>prev_prs_acc</code>	0.45	0.26	0.0	0.47	1.0
<code>first_pr_proj</code>	0.29	0.45	0	0.00	1
<code>followers</code>	3.1e2	1.5e3	0	41.00	3.3e4
<code>core_team</code>	0.33	0.47	0	0.00	1
<code>social_strength</code>	0.62	0.46	0.0	1.00	1.0
Patch-level characteristics					
<code>early_arrival</code>	0.50	0.50	0	0.50	1
<code>files_changed</code>	11.60	73.44	0	2.00	1.4e3
<code>loc</code>	6.0e2	5.0e3	0	17.00	9.9e4
<code>test_inclusion</code>	0.37	0.48	0	0.00	1
<code>issue_tag</code>	0.37	0.48	0	0.00	1
<code>text_len</code>	4.9e2	1.4e3	4	2.6e2	2.7e4
Review-level characteristics					
<code>revisions</code>	0.83	2.11	0	0.00	27
<code>comments</code>	5.18	11.84	0	2.00	3.3e2
<code>comments_inline</code>	1.55	5.42	0	0.00	1.1e2
<code>comments_pos</code>	0.17	0.27	0.0	0.00	1.0
<code>comments_neg</code>	0.04	0.13	0.0	0.00	1.0
<code>CI</code>	0.10	1.05	-1.0	0.00	3.0
<code>CLA</code>	-0.81	0.60	-1.0	-1.00	3.0
<code>CR</code>	-0.84	0.56	-1.0	-1.00	3.0

6.2.2 Statistical Analysis

Our goal is to explain the relationship (if any) between the selected factors and the binary response variable *accept*, which indicates whether a duplicate pull request has been accepted over its counterpart (1 for accepted and 0 for not accepted). We use conditional logistic regression [33], implemented in the *mclogit* package of **R**, to model the likelihood of duplicate pull requests being accepted based

on the matched pair samples in our dataset (*i.e.*, two paired duplicate pull requests in which one is merged and the other is rejected).

Similar to the analysis in Section 5.2, we add one level metrics at a time and build a model instead of building one model with all metrics at once. As a result, we compare the fit of three models: a) *Model 1*, which includes only the submitter-level variables, b) *Model 2*, which adds patch-level variables, and c) *Model 3*, which adds review-level variables. The variable transformation and multicollinearity control in the model construction process is similar to those in Section 5.2. Specifically, we remove three predictors (`prev_pullreqs_proj`, `prev_prs_acc_proj`, and `first_pr`) due to multicollinearity, which leaves us with 20 predictors, as shown in Table 13.

6.2.3 Analysis results

Overall, as shown in Table 14, both Model 2 and Model 3 have achieved remarkable performance given their high value of AUC [60]. Since Model 3 performs better than Model 2 (AUC: 0.890 vs 0.856) and they have obtained consistent variable effects, we discuss their effects based on Model 3.

As for the submitter-level predictors, only the predictor `prev_prs_acc` has a significant, positive effect. This means that duplicate pull requests submitted by experienced developers whose previous pull requests have a higher acceptance rate are more likely to be accepted. This is inline with the prior findings about general pull request evaluation [42], [47], [52], [116]. Perhaps surprisingly, the predictors relating to developers' standing (`core_team` and `followers`) and their social connection to the project (`social_strength`) are not significant by controlling for other confounds. Prior studies have shown that pull requests from the developers holding higher standing and having stronger social connection with the project have higher acceptance rates [93], [108]. However, our model does not achieve significant effects. Except for the bias on our dataset, we present an assumption that in the context of making a choice between duplicates, integrators' decision does not differentiate based on the identity of the submitter in order to ensure fairness within the community [19], [46]. This assumption deserves a further investigation.

For patch-level metrics, `early_arrival` is highly significant in the model. As expected, duplicate pull requests submitted earlier have a higher likelihood of being accepted. We can also observe that the predictor `loc` has a positive effect, which indicates that duplicate pull requests changing more LOCs are more likely to be accepted. This finding is opposite with the results in previous studies [93], [101] that large pull requests are less likely to be accepted. For a pair of duplicate pull requests, the large one might provide a more thorough solution or fix additional related issues compared to the small one, which increases the probability of acceptance under the same conditions. For example, there is a typical comment left by the integrator: *"Since the PR also contains some test cleanup, I'll merge that instead and close this one. But thank you for the efforts, it is much appreciated!!"*. In addition to `loc`, the predictor `inclusion_test` presents a significant, positive effect, which is similar to the effect on pull request acceptance in general [42], [93], [108].

Finally, we discuss the review-level metrics. The predictor `comments_inline` has a significant, positive effect. This indicates that duplicate pull requests receiving more inline comments have a higher chance of being accepted. Nevertheless, prior study [93] showed that pull requests with a high amount of discussion are less likely to be accepted. We argue that duplicate pull requests receiving more comments, especially inline comments, might mean that they have been reviewed and discussed more thoroughly than their counterparts. It requires less effort to be spent on the follow-up review if integrators choose the highly discussed duplicates. We notice that the predictor `revisions` has a positive effect. This indicates that duplicate pull requests revised more times are more likely to be accepted, which agrees with what was already found in prior study [47]. As for comment sentiment, unsurprisingly, duplicate pull requests receiving more positive comments than negative comments are more likely to be accepted. In terms of DevOps checking, as expected, the predictor `CI` has a strong, positive effect when its value is `success` or `pending` compared to when the value is `failure`. This means that duplicate pull requests have lower likelihood of being accepted if the CI test result is `failure`. Our result confirms the finding in the previous study [98], [108] that CI plays a key role in the process of pull request evaluation, even in the context of duplication. We do not achieve any result for other two DevOps tools (*i.e.*, CLA and CR), probably due to the imbalanced data (*i.e.*, most of pull requests are not checked by these tools). We plan to conduct further analysis focusing on these two kinds of tools in future work.

6.3 Manual inspection.

The regression analysis examines the correlation between several factors and integrators' choice between duplicate pull requests. It reveals what kind of duplicates are more or less likely to be accepted. We further investigate the exact reasons why integrators accept a duplicate pull request rather than its counterpart. This can also examine and verify the results of the regression analysis. To this end, we analyze the review comments of duplicate pull requests and perform a card sort [82] to gain insight into the common themes around integrators' choice between duplicates. The following sections present the card sorting process and the identified reasons for integrators' choice.

6.3.1 Card sorting

The card sorting analysis is conducted on 150 randomly selected duplicate pairs. This sample yields a 90% confidence level with a 6.28% error margin. This process includes the following three steps.

Card preparation: For each randomly selected duplicate pair, we read the dialogue and select all the comments expressing integrators' choice preference between duplicates. The selected comments are then recorded in a card.

Pair execution: For each card, two authors read the text and sort the card into an existing category. If the authors believe that the card does not belong to any of the existing categories, they create a new category for that card. The created category is labeled with a descriptive title to indicate

TABLE 14: Statistical models for the acceptance of duplicate pull requests

	Model 1			Model 2			Model 3		
	response: <i>accept</i> = 1			response: <i>accept</i> = 1			response: <i>accept</i> = 1		
	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.
$\log(\text{prev_pullreqs} + 0.5)$	-0.078	-1.832		-0.097	-1.987	*	-0.091	-1.738	
$\log(\text{prev_prs_acc} + 0.5)$	1.597	7.884	***	1.729	7.363	***	1.644	6.617	***
first_pr_proj TRUE	-0.206	-1.270		-0.169	-0.911		-0.173	-0.868	
$\log(\text{followers} + 0.5)$	0.061	1.674		0.062	1.477		0.064	1.415	
core_team TRUE	0.251	1.786		0.184	1.111		0.237	1.312	
$\log(\text{social_strength} + 0.5)$	0.383	2.590	**	0.217	1.295		0.283	1.538	
early_arrival TRUE	-	-	-	0.534	6.865	***	0.483	5.572	***
$\log(\text{loc} + 0.5)$	-	-	-	0.462	5.701	***	0.386	4.477	***
$\log(\text{files_changed} + 0.5)$	-	-	-	0.388	2.341	*	0.200	1.135	
test_inclusion TRUE	-	-	-	0.239	1.034		0.203	2.805	*
issue_tag TRUE	-	-	-	0.255	1.855		0.261	1.734	
$\log(\text{text_len} + 0.5)$	-	-	-	0.152	2.253	*	0.133	1.877	
$\log(\text{revisions} + 0.5)$	-	-	-	-	-	-	0.275	2.563	*
$\log(\text{comments} + 0.5)$	-	-	-	-	-	-	0.036	0.406	
$\log(\text{comments_inline} + 0.5)$	-	-	-	-	-	-	0.208	2.138	*
$\text{comments_pos} > \text{comments_neg}$ TRUE	-	-	-	-	-	-	0.460	3.315	***
CI Success	-	-	-	-	-	-	0.728	3.574	***
CI Pending	-	-	-	-	-	-	1.175	5.226	***
CLA Success	-	-	-	-	-	-	1.596	0.532	
CLA Pending	-	-	-	-	-	-	2.948	0.984	
CR Success	-	-	-	-	-	-	1.4e+01	0.022	
CR Pending	-	-	-	-	-	-	1.4e+01	0.022	
Akaike’s Information Criterion (AIC):		1411.52			1116.52			1014.19	
Bayesian’s Information Criteria (BIC):		1436.39			1171.23			1118.64	
Area Under the ROC Curve (AUC):		0.704			0.856			0.890	

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$










its theme. For a card citing multiple themes, more copies of the card are created for each cited theme. When the two authors disagree about the category of a specific card, they invite the other authors to discuss the discrepancy and vote on the card.

Final analysis: After all cards have been sorted, the two authors review each of the cards again to ensure the integrity of the emerged categories and resolve potential inclusive or redundant relations among categories. All categories are then further grouped into higher-level categories. Finally, to reduce the bias from the two authors, all authors of the paper review and agree on the final taxonomy of categories.

6.3.2 Reasons for integrators’ choice

As shown in Table 15, we find 8 reasons for integrators’ choice between duplicates, which can be classified in to two categories, *i.e.*, *technical assessment* and *non-technical consideration*. Note that the total frequency is greater than 150, because more than one reason might be cited in a decision-making discussion. In the following, we discuss each of reasons supported by examples.

TABLE 15: The taxonomy of the reasons for integrators’ choice between duplicates

	Reason	Frequency
Technical assessment	Higher quality	20 
	Correctness	11 
	Broader coverage	7 
	Pushed to proper branch	4 
	Inclusion of test code	3 
Non-technical consideration	First-come, First-served	30 
	Active response	3 
	Encouraging newcomers	1 
	No explanation	76 

Technical assessment.

Higher quality: If both of two duplicates have provided correct implementation, integrators are inclined to choose the one of higher quality. High quality has multiple manifestations in our analysis, such as less affected files (*e.g.*, “After

a short discussion internally, we are going to close this one in favour of merging #7666. This is nothing more than the fact that the other PR touches much less files”), and more outstanding performance (e.g., “Closing this in favor of #10373, which also contains performance improvements”). Given that higher-quality implementation tends to receive more positive review comments, this choice preference is consistent with the regression analysis result that duplicate pull requests receiving more positive comments are more likely to be accepted.

Correctness: The acceptance of duplicate pull requests fundamentally depends on the correctness of its implementation, as described by an integrator “Looks like #2716 fixed this though the quoting on that one is not 100% right and I prefer your solution... Would you like to rebase yours on top of head to use the ‘+=(...)’ operator?”. Since a failed status check indicates problematic implementation, this preference supports the finding in the regression analysis that duplicate pull requests are less likely to be accepted if the status is failure.

Broader coverage: Two pull requests might be partial duplicates in which one is the subset of the other one, as a reviewer pointed “This PR is a subset of PR #16031”. In such case, we observe that integrators prefer to choose the large one that has covered more issues. Considering that the broader coverage usually leads to more changed LOCs, this choice preference is reflected in the regression analysis that duplicate pull requests changing more LOCs are more likely to be accepted.

Pushed to the proper branch: It is common for OSS projects to follow a certain strategy on branch management. For example, the maintenance branch in many projects accepts only bug fixes, and new feature proposals are not accepted. Therefore, integrators accept duplicate pull requests pushed to the proper branch (e.g., “closing this PR as a duplicate of #5193 (the latter is sent against 2.0 which is the branch to target for bugfix)”).

Inclusion of test code: Most OSS project require contributors to provide necessary test codes for their modifications. Therefore, the existence of test code in the duplicate pull requests could help win some favor from integrators (e.g., “Thanks for the fix. Not sure we want to merge this or wait for #3907 which also fixes it and adds a regression test for all estimators”). This agrees with the result in the regression analysis.

Non-technical consideration.

First-come, first-served: Integrators may follow the “first-come, first-served” rule and accept duplicates arriving earlier than those arriving later (e.g., “Actually, I see that #28026 is at the head of the merge queue right now, so it will probably merge before this one, in which case we can just discard this PR” and “Thank you very much for opening the pull request @xxx. Sadly I have to close it nonetheless because someone already opened one with the same changes before: #5761”). This is consistent with the finding in the regression analysis that duplicates submitted earlier have a higher possibility of being accepted.

Active response: In the review process of a pull request, contributors are usually requested to update their pull requests until integrators are satisfied. Consequently, if a duplicate pull request has not been actively updated by its submitter, integrators might turn to its counterpart for which integrators have received active responses (e.g., “@xxx The original author of that PR hasn’t responded in a week. If you want to fix the tests in your PR, we could merge this one. Your call”).

Encouraging newcomers: New contributors act as an innovation source for new ideas and are essential to the survival of OSS projects [84]. Integrators have always tried to retain newcomers and hope that they become long-term contributors [112]. Consequently, duplicate pull requests submitted by newcomers might be accepted by integrators to encourage newcomers to make more contributions (e.g., “@xxx do you mind if we close this in favor of #12004 from a new contributor?”). However, we find very few instances of this choice preference. This is in line with the fact that the factors `first_pr_proj` and `core_team` show no significant effects in the regression model. This finding can indicate that when making a choice between duplicates, integrators care more about the pull requests than the role of their submitters.

No explanation. Integrators may make a choice without leaving any further explicit explanation (e.g., “Replaced by #5067” and “closing in favour of #10582”). For these cases, we examine the submission time of the involved duplicate pull requests. We find that most accepted pull requests are those that are submitted earlier. Therefore, a possible explanation for why integrators offer no explicit explanation is that they think the “first-come, first-served” rule is the default standard in duplication choice. In addition, integrators and contributors might have discussed the matter on other communication media or face-to-face outside GitHub, and as a result, integrators simply make a choice without more explanation.

6.3.3 Complementary Investigation: Beyond rejection

In examining integrators’ comments, we also find that, sometimes, making a decision between duplicates is more complex than simple acceptance and rejection. Even though integrators give their preference to one duplicate pull request, they might not directly and completely abandon the other one. In the following, we present three instructive scenarios where integrators make a decision beyond simple close in dealing with the rejected duplicates.

Supplementary. Although it has been decided to accept one duplicate pull request instead of another one, some integrators would examine what could be reused from the rejected one to enhance the accepted one. For example, although `rails/rails/#20697` was preferred over `rails/rails/#20050`, the author of `rails/rails/#20697` was asked to reuse the test code from `rails/rails/#20050` (“I prefer this behavior (and its simpler implementation) Can you please cherry-pick the test from #20050”). In such cases, integrators usually also gave credits to the author of the rejected pull request for the incorporated codes (“along with adding an changelog entry listing both authors” and “Can you credit him in the commit as well after you squash and rebase”).

Reopen and backup. Some rejected duplicate pull requests were later reopened as useful backups, since their counterparts that were previously preferred could not be merged finally. For example, *facebook/react/#5236* was closed at first because there was already a duplicate pull request *facebook/react/#5220*. But later, *facebook/react/#5236* was reopened and merged due to the inactivity of *facebook/react/#5220*, as the integrator said: “Other PR didn’t get updated today and the failures were annoying on new PRs”.

Collaboration. Integrators might ask the author of the rejected duplicate pull request to improve the preferred one together (e.g., “Could you jump on #3082 and help us review it?” and “Please work with @xxx to test his code once he fixes the merge conflicts”). We also found that some of the authors were willing to offer their help (e.g., “Made a minor note on the other PR to include one of the improvements” and “Looks like they took my advice and used `proc_terminate()` so another package was not needed. Thanks for catching duplicate, but at least it was not a waste. :)”)

RQ3: Pull requests with accurate and high-quality implementation, broad coverage, necessary test code, high maturity, and deep discussion are more likely to be accepted. However, integrators also make a choice based on non-technical considerations, e.g., they may accept pull requests to respect the arrival order and active response. For the rejected duplicates, integrators might try to maximize their value, e.g., cherry-picking the useful code.

7 DISCUSSION

Based on our analysis results and findings, we now provide additional discussion and propose recommendations and implications for OSS practitioners.

7.1 Main findings

7.1.1 Awareness breakdown

Maintaining awareness in global distributed development is a significant concern [43], [92]. Developers need to pursue “an understanding of the activities of others, which provides a context for your own activity” [38]. As the most popular collaborative development platform, GitHub has centralized information about project statuses and developer activities and made them transparent and visible [37], which help developers to maintain awareness with less effort [48]. However, awareness breakdown still occurs and results in duplicate work. Our findings about the specific contexts where duplicates are produced, as shown in Section 5.1, highlight three mismatches leading to awareness breakdown.

A mismatch between awareness requirements and actual activities. In most community-based OSS projects, developers are self-organized [26], [34], and are allowed to work on any part of the code according to individual’s interest and time [43], [54]. Awareness requirements arise as a response to developers’ free and spontaneous activities. Whenever a developer decides to get engaged in a task, s/he should ensure that no other developers have worked on the same task. However, our findings show that some contributors

lack sufficient effort investment in awareness activities (Section 5.1: *Not searching for existing work*, *Overlooking linked pull requests*, and *Missing existing claim*). We assume that this is due to the volunteer nature of OSS participation. For some developers, especially the casual contributors and one time contributors, a major motivation to make contributions is to “scratch their own itch” [56], [69]. When they encounter a problem, they code a patch to fix it and send the patch back to the community. Some of them even do not care about the final outcome of their pull requests [85]. It might be harder to get them to spend more time to maintain awareness of other developers. Automatic awareness tools can mitigate this problem. Prior research has proposed to automatically detect duplicates at pull request submission [57], [73] and identify ongoing features from forks [113]. Furthermore, we advocate for future research on seamlessly integrating awareness tools to developers’ development environment and designing intelligent and non-intrusive notification mechanism.

A mismatch between awareness mechanisms and actual demands. Currently, GitHub provides developers with a wide range of mechanisms, e.g., following developers and watching projects [37], to maintain a general awareness about project status. However, developers can be overwhelmed with a large-scale of incoming events in popular projects (Section 5.1: *Missing notifications*). It is also impractical for developers to always maintain overall awareness of a project due to multitasking [95] and turnover [58]. Usually, developers need to obtain on-demand awareness around a specific task whenever deciding to submit a pull request, i.e., gathering task-centric information to figure out people interested in the same task. Currently, the main mechanisms to meet this demand are querying issue and pull request list and reading through the discussion history. As mentioned in Section 5.1, the support by these mechanisms is not as adequate as expected due to information mixture (Section 5.1: *Overlooking linked pull requests* and *Missing existing claims*) and other technical problems (Section 5.1: *Disappointing search functionality* and *Diversity of natural language usages*). Awareness mechanisms would be most useful if they can fulfil developers’ actual demands in maintaining awareness.

A mismatch between awareness maintenance and actual information exchange. Maintaining awareness is bidirectional. Intuitively, it means that developers need to *gather* external information to stay aware of others’ activities, with the hope that *I do not duplicate others’ work*. But from a global perspective, it also means developers should actively *share* their personal information that can be gathered by others, with the hope that *others do not duplicate my work*. Our findings show that some developers do not timely announce their plans (Section 5.1: *Implementing without claiming first*) and link their work to the associated issue (Section 5.1: *Lack of links*). This hinders other developers’ ability to gather adequate contextual information. Although prior work [18], [29], [92] has extensively studied on how to help developers track work and get information, more research attention should be paid to encouraging developers to share information. For example, it would be interesting to investigate whether developers’ willingness to share information is affected by the characteristics of collaboration mechanisms

and communication tools.

Obviously, awareness tools are important for OSS developers to stay aware of each other. However, no tool or mechanism can prevent all awareness breakdown entirely. A better understanding of the importance of group awareness and better use of available technologies can help developers ensure that their individual contributions do not cause accidental duplicates.

7.1.2 The paradox of duplicates

Generally speaking, both project integrators and contributors hope to prevent duplicate pull requests, because duplicates can waste their time and effort, as shown in Section 4. This is also reflected in their comments, e.g., *“it probably makes sense to just center around a single effort”*, *“No need to do the same thing in two PRs”*, and *“Oops! Sorry, did not mean to double up”*. However, when duplicates are already produced, potential value might be mined from them as shown in Section 6.3.3. From our findings, we notice two interesting paradoxes of duplicates.

Redundancy vs. alternative. In many cases, duplicate pull requests change pretty much the same codes, which only bring unnecessary redundancy. While in some cases, duplicates implemented in different approaches provide alternative solutions, as a developer put it: *“The pull requests are different, so maybe it is good there are two”*. In such cases, project integrators have a higher chance to accept a better patch. However, this comes at a price. Integrators have to invest more time to compare the details of duplicates in order to clearly disclose the difference between them. Ensuring that their effort is not wasted in coping with duplicates, but maximizing the disclosure and adoption of additional value provided by each duplicate, is a trade-off integrators should be aware of.

Competition vs. collaboration. At first sight, authors of duplicate pull requests face a competition in getting their own patches accepted. For example, one contributor tried to persuade integrators to choose his pull request rather than another one: *“Pick me! Pick me! I was first! :)”*. Nevertheless, we found some cases where the authors of duplicates worked together towards a better patch by means of mutual assessment and negotiated incorporation, as shown in Section 6.3.3. According to developers, the collaboration is also an opportunity for both the authors to learn from each other’s strength (*“I looked at some awesome code that @xxx wrote to fix this issue and it was so simple, I just did not fully understand the issue I was fixing”*). Standing for their own patches, but seeking for collaboration and learning, is a trade-off the authors of duplicates should be aware of.

7.1.3 Decision-making in Either/Or contexts

In the general context of pull request evaluation, integrators are answering a Yes/No question *“whether to accept this pull request?”*, and the considered factors are mainly relating to individual pull requests. While in the context of making a choice between duplicates, integrators are answering an Either/Or question *“whether to choose this duplicate pull request or the other one?”*, and integrators would evaluate the duplicates from a comparative perspective. Based on our findings about integrators’ preference between duplicates,

as presented in Section 6, we infer two prominent characteristics of integrators’ decisions in the Either/Or contexts.

Choose the patch, not the author: We find that when making decisions between duplicates, integrators have a preference to consider patch-level metrics (e.g., arrival order) and review-level metrics (e.g., CI test results) instead of submitter-level metrics (e.g., the submitter’s identity). Compared to submitter-level metrics, patch-level and review-level metrics are more objective evidence. While the submitter’s identity can be used to make inferences about the quality and trustworthiness of a pull request [42], [93], decisions made on the basis of objective evidence might look more fair and rational in the context of selection between duplicates. The benefits of such decision strategy include ensuring the fairness within the communities [19], [46] and eliminating integrators’ pressure of explaining the rejection of duplicates [42].

Invested effort matters: We find that integrators prefer duplicate pull requests that have been highly discussed and revised a couple of times. While a higher number of comments and revisions might indicate that a pull request was not perfect and integrators have requested changes to update it, it also reflects that the pull request has been thoroughly reviewed and improved, and both integrators and the submitter have invested considerable effort. The invested effort on one duplicate pull request cannot be “transferred” to its counterpart because duplicate pull requests might have different implementation details and each of them has to be carefully reviewed. Given the facts that time is the top challenge faced by integrators [42] and that asking for more work from contributors to improve their work might be difficult [42], [85], choosing the thoroughly discussed and revised duplicate pull requests might be a cost-efficient and safe decision.

7.2 Suggestions for contributors

To avoid unintentional duplicate pull requests, contributors may follow a set of best contributing practices when they are involved in the pull-based development model.

Adequate checking: Many duplicates were produced because contributors did not conduct adequate checking to make sure that no one else was working on the same thing (Section 5.1: *Not searching for existing work*, *Overlooking linked pull requests*, and *Missing existing claims*). We recommend that contributors should perform at least three kinds of checking before starting their work: *i)* reading through the whole discussion of an issue and checking whether anyone has claimed the issue; *ii)* examining each of the pull requests linked to an issue and checking whether any of them is an ongoing work to solve the issue; and *iii)* performing searches with different keywords against open and closed pull requests and issues, and carefully checking where similar work already exists.

Timely completion: Quite a number of OSS developers contribute to a project at their spare time, and some of them even switch between multiple tasks. As a result, it might be difficult for them to complete an individual task in a timely fashion. However, we still suggest that contributors should quickly accomplish each work in proper order, e.g.,

one item at a time, to shorten their local duration. This can make their work publicly visible earlier, which can, to some extent, prevent others from submitting duplicates (Section 5.1: *Overlong local work*).

Precise context: Providing complete and clear textual information for submitted pull requests is helpful for other contributors to retrieve these pull requests and acquire an accurate and comprehensive understanding of them (Section 5.1: *Diversity of natural language usage*). In addition, if a pull request is solving a tracked issue, adding the issue reference in the pull request description, e.g., “fix #[issue_number]”, can avoid some duplicates because of the increased degree of awareness. (Section 5.1: *Lack of links*).

Early declaration: Zhou *et al.* [114] already suggested that claiming an issue upfront is associated with a lower chance of redundant work. In our study, we find several actual instances of duplicates where integrators clearly pointed out the contributors should claim the issues first and then implement the patches (e.g., “@xxx, btw, it is a good idea to comment on an issue when you start working on it, so we can coordinate better and avoid duplication of effort”). We would like to emphasize again the importance of early declaration which should become a best practice developers can follow in OSS collaborative development. Compared with late report, early declaration can timely broadcast contributors’ intention to the community to get the attention of interested parties, so that they can avoid some accidental duplicate work (Section 5.1: *Implementing without claiming first*).

Argue for their patches: As shown in Section 6, various factors can be examined when integrators make decisions between duplicates. The authors of duplicates should actively argue for their own pull requests by explicitly stating the strength of their patches, especially if they have proposed a different approach and provided additional benefits. They can also review each of other’s patch and discuss the difference before waiting for an official statement from integrators. This can provide a solid basis for integrators to make informed decisions about which duplicate should be accepted. Moreover, if the value of a duplicate pull request has been explicitly stated, even it is finally closed, its useful part has a higher chance to be noticed and cherry-picked by integrators, as shown in Section 6.3.3.

7.3 Suggestions for core team

The core team of an OSS project, acting as the integrator and maintainer of the project, is responsible for establishing contribution standards and coordinating contributors’ development. To achieve the long-term and continuous survival of the project, the core team may also follow some best practices.

Evident guidelines: Although most projects have warned contributors not to submit duplicate issues and pull requests, the advice are usually too general. We suggest projects to make the advice more visible, specific, and easy-to-follow. For example, projects can use a section to list the typical contexts where duplicates occur, as presented in Section 5.1, and itemize the specific actions should be taken to avoid duplicates, as we have suggested for contributors (Section 7.2: *Adequate checking*).

Explaining decisions: Integrators must make a choice between duplicate pull requests, which means that they have to reject someone. For contributors whose pull requests have been rejected, they might be pleased to get feedback and explanation about why their work has been rejected rather than simply closing their pull requests. However, we observed nearly 50% of our qualitative samples where decisions were made without any explanation (as shown in Table 15). Even worse, we identified that the rough explanation (e.g., “Thanks for your PR but this fix is already merged in #20610”) would be likely to make the contributor upset (“‘already’ implies I submitted my PR later than that, rather than nearly a year earlier ;) But at least it’s fixed”). In that case, the integrator had to give an additional apology (“sorry, sometimes a PR falls in the cracks and a newer one gets the attention. We have improved the process in hopes to avoid this but we still have a big backlog in which these things are present”) to mitigate the negative effect. In the future, a careful analysis should be designed to examine the effectiveness of this suggestion based on controlled experiments.

7.4 Suggestions for design of platforms

Online collaborating platforms such as GitHub have designed and provided numerous mechanisms and tools to support OSS development. However, the practical problem of duplicate contributions proves that the platforms need to be improved.

Claim button: In order to make it more efficient for developers to maintain awareness of each other, we envision a new mechanism called *Claim* which is described as follows. For a GitHub issue, each interested developer can click the *Claim* button on the issue page to claim that s/he is going to work on the issue. The usernames of all claimers are listed together below the *Claim* button. Every time the *Claim* button is clicked, an empty pull request is automatically generated and linked to the claimer’s username in the issue claimer list. Moreover, claimers have a chance to report their plans about how to fix the issue in the input box displayed when the *Claim* button is clicked. The reported plans would be used to describe the empty pull request. Subsequently, claimers perform updates of the empty pull request until they produce a complete patch. All important updates on the empty pull request, e.g., new commits pushed, would be displayed in the claimer list. On the one hand, this mechanism makes it more convenient for developers to share their intentions and activities through just clicking a button. On the other hand, developers can efficiently catch and track other developers’ intentions and activities by simply checking the issue claimer list.

Duplicate detection: As contributors complained, e.g., “... I wish there has been some automated method to detect pending PR per file basis. This could save lot of work duplicacy. ...”, or “It’s strange that GitHub isn’t complaining about this, because it’s an exact dup of #5131 which was merged already”, an automatic detection tool of duplicates is missing in GitHub. Such a tool can help integrators detect duplicates in a timely manner and prevent them spending resources on the redundant effort of evaluating duplicates separately. Therefore, GitHub can learn from Stack Overflow and Bugzilla to recommend

similar work when developers are creating pull requests by utilizing various similarity measures, *e.g.*, title and code changes. The features discussed in Section 5.2.1 can also be integrated to enhance the recommendation system.

Reference reminder: Since developers might overlook linked pull requests to issues (Section 5.1: *Overlooking linked pull requests*), platforms can actively remind developers of existing pull requests linked to the same issue at pull request submission time. The goal of this functionality is similar to that of the duplicate detection tool. However, it can be implemented in a more straightforward way. For example, whenever developers add an issue reference in filling a pull request, a pop-up box can be displayed next to the issue reference to list the existing pull requests linked to that issue.

Duplicate comparison: As discussed in Section 6, when integrators make a choice between duplicate pull requests, they consider several factors. Platforms can support duplicate comparison to make the selection process more efficient. For example, platforms can automatically extract several features of compared duplicates, *e.g.*, inclusion of test codes and the contributor’s experience, and display these features in a comparison format to clearly show the difference between duplicate pull requests and speed up the selection process.

Online incorporation: As presented in Section 6.3.3, integrators sometimes prefer to incorporate one duplicate pull request into the other one to promote patch thoroughness. Currently, the typical way to incorporate a pull request PR_i into another pull request PR_j is as follows: *i)* adding the head branch of PR_i as a remote branch in the corresponding local repository of PR_j , *ii)* fetching the remote branch to the local repository, *iii)* cherry-picking the needed commits from or rebase onto the remote branch, and *iv)* updating PR_j by synchronizing the changes from local repository to the head branch of PR_j . Developers might also need to update the commit message or project changelog to give credit for the incorporated code. The whole incorporation process can be too complex for newcomers to undertake. Moreover, this process seems to be tedious for incorporating trivial changes. GitHub can support online incorporation of duplicate pull requests. For example, it can allow developers to pick the needed code by clicking buttons in the UI, and the credit is given to the picked code by automatically updating the commit message and changelog.

8 THREATS TO VALIDITY

In this section, we discuss threats to construct validity, internal validity and external validity, which may affect the results of our study.

Construct Validity: In the definition of sequential relationships between two duplicates, two time points are critical, *i.e.*, $T\text{-Creation}$ and $T\text{-Evaluation}$, which stand for the starting times of local work and online evaluation, respectively. In the paper, we set $T\text{-Evaluation}$ as the submission time of pull request, in accordance with its definition. Nevertheless, we cannot obtain the exact value of $T\text{-Creation}$ because there is no information recording when a contributor starts

local work. We set $T\text{-Creation}$ as the creation time of the first commit contained in a pull request. As a result, the observed value of $T\text{-Creation}$ is actually later than its real value because it can be certain that the contributor must first start the local work and then later submit the first commit. It is possible that we introduce some bias to our quantitative study when we set $T\text{-Creation}$ to the creation time of the first commit. For duplicate pairs of overlapping or inclusive relations, the bias does not matter much because they already intersect with each other. However, the bias for duplicate pairs of exclusive relations needs careful attention since inaccurate value of $T\text{-Creation}_{dup}$ may affect whether the relation is exclusive. Indeed, in our quantitative study of the exclusive interval (Table 8), we find that the majority of duplicate pairs of exclusive relations have relatively long intervals, which means that a minor shift in the value of $T\text{-Creation}_{dup}$ is unlikely to affect the original relation. Therefore, setting $T\text{-Creation}$ as the creation time of the first commit is acceptable in practice.

Internal Validity: In the manual analysis of integrators’ choice between duplicate pull requests, we target a sampled subset of duplicate pull requests. It is possible that we have missed some other cases that are not in the sampled subset. However, the items in the subset are randomly selected, and the sample size is of high confidence level, as described in Section 6.3. Therefore, missed cases (if any), accounting for a very small proportion of the whole, would not significantly change our findings.

External Validity: The threat to external validity relates to the generalizability of our findings. To mitigate this threat, our empirical study was conducted on 26 projects hosted on GitHub, covering a diversity of programming languages and application domains. However, it is still a small sample given that 100 million repositories [1] have been hosted on GitHub, let alone there are other social coding sites such as GitLab and BitBucket. In the future, we plan to extend our study by including more projects from jointCloud [99] development platforms.

9 CONCLUSION

In this study, we investigated the problem of duplicate contributions in the context of pull-based distributed development. The goal of our study is to better understand the influences of duplicate pull requests during collaborative development, the context in which duplicate pull requests occur, and the alternative preference of integrator between duplicate pull requests. We conducted an empirical study on 26 GitHub projects to achieve the goal. We found that duplicate pull requests slow down the review process and require more reviewers for extended discussions. We observed that the inappropriateness of OSS contributors’ work patterns (*e.g.*, not checking for existing work) and the shortcomings of their collaboration environment (*e.g.*, unawareness of parallel work) would result in duplicates. We also observed that duplicate pull requests are significantly different from non-duplicate pull requests in terms of project-level characteristics (*e.g.*, area hotness and number of active core team members), submitter-level characteristics (*e.g.*, experience and social connection to project), and patch-level characteristics (*e.g.*, change type and issue visibility).

We found that duplicate pull requests with accurate and high-quality implementation, broad coverage, necessary test codes, high maturity, and deep discussion, are more likely to be accepted. We also found that integrators might make a choice based on non-technical considerations, *e.g.*, they may accept pull requests to respect arrival order and active response.

Based on the findings we recommend that OSS contributors should always perform sufficient verification against existing work before they start working on a task. Contributors are expected to declare their intentions as soon as possible and prepare their work with complete related information to make their work highly visible early on. Integrators should provide contributors with visible and detailed guidelines on how to avoid duplicated work. Social coding platforms are expected to enhance the awareness mechanisms in order to make it more effective and efficient for developers to stay aware of each other. It is also meaningful to provide practical service and tools to support automatic identification of duplicates, visualized comparison between duplicates, *etc.*

Last but not least, our findings point to several future research directions. Researchers can design awareness tools to increase developers' awareness of others activities. Such tools not only help prevent duplicate effort on the same tasks but also have the potential functionality to link related contributors for better coordination. Moreover, we think it is meaningful to investigate how integrators' practices in managing the contributors' conflicts affect contributors' continuous participation.

ACKNOWLEDGMENTS

This work was supported by National Grand R&D Plan (Grant No. 2018AAA0102304) and National Natural Science Foundation of China (Grant No. 61702534).

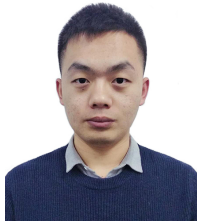
REFERENCES

- [1] About github. <https://github.com/about>. Accessed: 2019-10-19.
- [2] About github issue. <http://help.github.com/en/articles/about-issues>. Accessed: 2019-10-19.
- [3] Bugzilla. <http://www.bugzilla.org>. Accessed: 2019-10-19.
- [4] Code climate. <https://codeclimate.com>. Accessed: 2019-10-19.
- [5] Contributing to scikit-learn. <https://scikit-learn.org/dev/developers/contributing.html>. Accessed: 2020-6-11.
- [6] Contributing to the ansible documentation. https://docs.ansible.com/ansible/devel/community/documentation_contributions.html#community-documentation-contributions. Accessed: 2020-6-11.
- [7] The duppr dataset. <https://github.com/whystar/MSR2018-DupPR>. Accessed: 2019-10-19.
- [8] Get the combined status for a specific ref. <https://developer.github.com/v3/repos/statuses/#get-the-combined-status-for-a-specific-ref>. Accessed: 2020-05-13.
- [9] Github. <http://github.com>. Accessed: 2019-10-19.
- [10] Gitlab. <http://gitlab.com>. Accessed: 2019-10-19.
- [11] Good first issues in the project pandas. <https://github.com/pandas-dev/pandas/labels/good%20first%20issue>. Accessed: 2020-6-11.
- [12] How should duplicate questions be handled? <https://meta.stackexchange.com/questions/10841/how-should-duplicate-questions-be-handled>. Accessed: 2019-10-19.
- [13] Stack overflow. <http://stackoverflow.com>. Accessed: 2019-10-19.
- [14] Travis-ci. <https://www.travis-ci.org>. Accessed: 2019-10-19.
- [15] Understanding the github flow. <https://guides.github.com/introduction/flow/>. Accessed: 2020-8-8.
- [16] Bram Adams, Bram Adams, Bram Adams, and Marco Ortu. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In *Proceedings of the 11th working conference on mining software repositories*, pages 262–271, 2014.
- [17] Muhammad Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. Mining duplicate questions in stack overflow. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 402–412. ACM, 2016.
- [18] Ritu Arora, Sanjay Goel, and Ravi Kant Mittal. Supporting collaborative software development over github. *Software: Practice and Experience*, 47(10):1393–1416, 2017.
- [19] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. The secret life of patches: A firefox case study. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, pages 447–455. IEEE, 2012.
- [20] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):1–28, 2015.
- [21] Andrew Begel, Jan Bosch, and Margaret Anne Storey. Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder. *IEEE Software*, 30(1):52–66, 2013.
- [22] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. In *Proceedings of 2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 356–367, 2017.
- [23] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society*, 57(1):289–300, 1995.
- [24] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful really? In *Proceedings of 2008 IEEE International Conference on Software Maintenance*, pages 337–345. IEEE, 2008.
- [25] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in oss projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 26. IEEE Computer Society, 2007.
- [26] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 24–35. ACM, 2008.
- [27] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution*, pages 334–344. IEEE, 2016.
- [28] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Chris Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development as well as industrial practice at microsoft. *IEEE Transactions on Software Engineering*, pages 1–1, 2016.
- [29] Fabio Calefato and Filippo Lanubile. Socialcde: a social awareness tool for global software teams. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 587–590, 2013.
- [30] Fabio Calefato, Filippo Lanubile, Federico Maiorano, and Nicole Novielli. Sentiment polarity detection for software development. *Empirical Software Engineering*, (3):1–31, 2017.
- [31] Scott Chacon and Ben Straub. *Pro Git (Second Edition)*. Apress, 2018.
- [32] Patricia Cohen, Stephen G West, and Leona S Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Psychology Press, 2014.
- [33] Margaret A Connolly and Kung-Yee Liang. Conditional logistic regression models for correlated binary data. *Biometrika*, 75(3):501–506, 1988.
- [34] Kevin Crowston, Kangning Wei, Qing Li, U Yeliz Eseryel, and James Howison. Coordination of free/libre open source software development. 2005.
- [35] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. Core and periphery in free/libre and open source software team communications. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, volume 6, pages 118a–118a. IEEE, 2006.

- [36] Laura Dabbish, Colleen Stuart, Jason Tsay, and James Herbsleb. Leveraging transparency. *IEEE software*, 30(1):37–43, 2013.
- [37] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.
- [38] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, 1992.
- [39] Denae Ford, Mahnaz Behroozi, Alexander Serebrenik, and Chris Parnin. Beyond the code itself: how programmers really look at pull requests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pages 51–60. IEEE, 2019.
- [40] Georgios Gousios, Martin Pinzger, and Arie Van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.
- [41] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor’s perspective. In *Proceedings of the 38th International Conference on Software Engineering*, pages 285–296. IEEE, 2016.
- [42] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator’s perspective. In *Proceedings of the 37th International Conference on Software Engineering*, pages 358–368. IEEE, 2015.
- [43] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 72–81. ACM, 2004.
- [44] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.
- [45] Abram Hindle, Michael W Godfrey, and Richard C Holt. Release pattern discovery: A case study of database systems. In *Proceedings of 2007 IEEE International Conference on Software Maintenance*, pages 285–294. IEEE, 2007.
- [46] Chris Jensen and Walt Scacchi. Role migration and advancement processes in ossd projects: A comparative case study. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007.
- [47] Yujuan Jiang, Bram Adams, and Daniel M German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 101–110. IEEE Press, 2013.
- [48] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M. German. Open source-style collaborative development practices in commercial projects using github. In *Proceedings of 2015 International Conference on Software Engineering*, 2015.
- [49] Yvonne Kammerer and Peter Gerjets. The role of search result position and source trustworthiness in the selection of web search results when using a list or a grid interface. *International Journal of Human-Computer Interaction*, 30(3):177–191.
- [50] Frank Konietzke, Ludwig A Hothorn, and Edgar Brunner. Rank-based multiple test procedures and simultaneous confidence intervals. *Electronic Journal of Statistics*, 6:738–759, 2012.
- [51] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: how developers see it. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering*, pages 1028–1038. IEEE, 2016.
- [52] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart De Water. 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*, pages 124–133, 2018.
- [53] Karim R Lakhani and Eric Von Hippel. How open source software works: free user-to-user assistance. In *Produktentwicklung mit virtuellen Communities*, pages 303–339. Springer, 2004.
- [54] Karim R Lakhani and Robert G Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. 2003.
- [55] Alina Lazar, Sarah Ritchey, and Bonita Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 308–311. ACM, 2014.
- [56] Amanda Lee, Jeffrey C Carver, and Amiangshu Bosu. Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: a survey. In *Proceedings of the 39th International Conference on Software Engineering*, pages 187–197. IEEE Press, 2017.
- [57] Zhixing Li, Gang Yin, Yue Yu, Tao Wang, and Huaimin Wang. Detecting duplicate pull-requests in github. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, page 20. ACM, 2017.
- [58] Bin Lin, Gregorio Robles, and Alexander Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *Proceedings of the 12th International Conference on Global Software Engineering*, pages 66–75. IEEE, 2017.
- [59] Jeffrey D. Long, Feng Du, and Norman Cliff. *Ordinal Analysis of Behavioral Data*. John Wiley & Sons, Inc., 2003.
- [60] Charles E Metz. Basic principles of roc analysis. In *Seminars in nuclear medicine*, volume 8, pages 283–298. Elsevier, 1978.
- [61] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.
- [62] Yuji Mizobuchi and Kuniharu Takayama. Two improvements to detect duplicates in stack overflow. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, pages 563–564. IEEE, 2017.
- [63] Audris Mockus and Lawrence G Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 2000 International Conference on Software Maintenance*, pages 120–130, 2000.
- [64] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85. ACM, 2002.
- [65] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. ACM, 2012.
- [66] Nicole Novielli, Fabio Calefato, Davide Dongiovanni, Daniela Girardi, and Filippo Lanubile. Can we use se-specific sentiment analysis tools in a cross-platform setting? *arXiv preprint arXiv:2004.00300*, 2020.
- [67] Jason W Osborne and Amy Overbay. The power of outliers (and why researchers should always check for them). *Practical assessment, research & evaluation*, 9(6):1–12, 2004.
- [68] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 2013 International Conference on Software Engineering*, 2013.
- [69] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. More common than you think: An in-depth study of casual contributors. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 112–123. IEEE, 2016.
- [70] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocchi, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. Supporting software developers with a holistic recommender system. In *Proceedings of the 38th International Conference on Software Engineering*, 2017.
- [71] Mohammad Masudur Rahman and Chanchal K Roy. An insight into the pull requests of github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 364–367, 2014.
- [72] Fred Ramsey and Daniel Schafer. *The statistical sleuth: a course in methods of data analysis*. Cengage Learning, 2012.
- [73] Luyao Ren, Shurui Zhou, Christian Kästner, and Andrzej Wasowski. Identifying redundancies in fork-based development. In *Proceedings of 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 230–241. IEEE, 2019.
- [74] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.
- [75] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35, 2014.
- [76] Peter C Rigby, Daniel M German, and Margaretanne Storey. Open

- source software peer review practices: a case study of the apache server. pages 541–550, 2008.
- [77] Peter C Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 541–550. IEEE, 2011.
 - [78] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, pages 499–510. IEEE Computer Society, 2007.
 - [79] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.
 - [80] Jyoti Sheoran, Kelly Blincoe, Eirini Kalliamvakou, Daniela Damian, and Jordan Ell. Understanding “watchers” on github. In *Proceedings of the 11th working conference on mining software repositories*, pages 336–339, 2014.
 - [81] Rodrigo Souza and Bruno C Da Silva. Sentiment analysis of travis ci builds. In *Proceedings of 2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 459–462, 2017.
 - [82] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.
 - [83] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 1379–1392. ACM, 2015.
 - [84] Igor Steinmacher, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa. Overcoming open source project entry barriers with a portal for newcomers. In *Proceedings of the 38th International Conference on Software Engineering*, pages 273–284. ACM, 2016.
 - [85] Igor Steinmacher, Gustavo Pinto, Igor Scaliante Wiese, and Marco Aurélio Gerosa. Almost there: A study on quasi-contributors in open-source software projects. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering*, pages 256–266. IEEE, 2018.
 - [86] Klaas-Jan Stol and Brian Fitzgerald. Two’s company, three’s a crowd: a case study of crowdsourcing software development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 187–198. ACM, 2014.
 - [87] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society, 2011.
 - [88] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 45–54. ACM, 2010.
 - [89] Yida Tao, Donggyun Han, and Sunghun Kim. Writing acceptable patches: An empirical study of open source project patches. In *Proceedings of 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 271–280. IEEE, 2014.
 - [90] Patanamom Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th International Conference on Software Engineering*, 2017.
 - [91] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web?: Nier track. In *Proceedings of the 2011 33rd International Conference on Software Engineering*, pages 804–807. IEEE, 2011.
 - [92] Christoph Treude and Margaret-Anne Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 365–374, 2010.
 - [93] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, pages 356–366. ACM, 2014.
 - [94] Jason Tsay, Laura Dabbish, and James Herbsleb. 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*, pages 144–154. ACM, 2014.
 - [95] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. The sky is not the limit: multitasking across github projects. In *Proceedings of the 38th International Conference on Software Engineering*, pages 994–1005, 2016.
 - [96] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark GJ van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. Gender and tenure diversity in github teams. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, pages 3789–3798, 2015.
 - [97] Bogdan Vasilescu, Alexander Serebrenik, Mathieu Goeminne, and Tom Mens. On the variation and specialisation of workload a case study of the gnome ecosystem community. *Empirical Software Engineering*, 19(4):955–1008, 2014.
 - [98] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.
 - [99] Huaimin Wang, Peichang Shi, and Yiming Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.
 - [100] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering*, pages 461–470. ACM, 2008.
 - [101] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76. ACM, 2008.
 - [102] Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A. Gerosa. The power of bots: Characterizing and understanding bots in oss projects. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW), November 2018.
 - [103] Joel West and Scott Gallagher. Challenges of open innovation: the paradox of firm investment in open-source software. *R&D Management*, 36(3):319–331, 2006.
 - [104] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. A conceptual replication of continuous integration pain points in the context of travis ci. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 647–658. ACM, 2019.
 - [105] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
 - [106] Yue Yu, Zhixing Li, Gang Yin, Tao Wang, and Huaimin Wang. A dataset of duplicate pull-requests in github. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 22–25. ACM, 2018.
 - [107] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 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.
 - [108] Yue Yu, Gang Yin, Tao Wang, Cheng Yang, and Huaimin Wang. Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences*, 59(8):080104, 2016.
 - [109] Wei Emma Zhang, Quan Z Sheng, Jey Han Lau, and Ermyas Abebe. Detecting duplicate posts in programming qa communities via latent semantics and association rules. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1221–1229. International World Wide Web Conferences Steering Committee, 2017.
 - [110] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One size does not fit all: an empirical study of containerized continuous deployment workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 295–306. ACM, 2018.
 - [111] Yun Zhang, David Lo, Xin Xia, and Jian-Ling Sun. Multi-factor duplicate question detection in stack overflow. *Journal of Computer Science and Technology*, 30(5):981–997, 2015.

- [112] Minghui Zhou and Audris Mockus. What make long term contributors: Willingness and opportunity in oss community. In *Proceedings of the 34th International Conference on Software Engineering*, pages 518–528. IEEE Press, 2012.
- [113] Shurui Zhou, Stefan Stanciulescu, Olaf Lebenich, Yingfei Xiong, Andrzej Wasowski, and Christian Kastner. Identifying features in forks. In *Proceedings of the 39th International Conference on Software Engineering*, pages 105–116, 2018.
- [114] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. What the fork: A study of inefficient and efficient forking practices in social coding. In *Proceedings of the 2019 27th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2019.
- [115] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Effectiveness of code contribution: From patch-based to pull-request-based tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 871–882. ACM, 2016.
- [116] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. *Empirical Software Engineering*, 24(6):3871–3903, 2019.



Zhixing Li is a PH.D. candidate in Software Engineering at National University of Defense Technology (NUDT). He received his Master degree in compute science from NUDT and his Bachelor degree from Chongqing University. His research goals are centered around the idea of making the open source collaboration more efficient and effective by investigating the challenges faced by open source communities and designing smarter collaboration mechanisms and tools.



Yue Yu is an associate professor in the College of Computer at National University of Defense Technology (NUDT). He received his PH.D. degree in Computer Science from NUDT in 2016. He has won Outstanding Ph.D. Thesis Award from Hunan Province. His research findings have been published on ICSE, FSE, ASE, ESEM etc. His current research interests include software engineering, data mining and computer-supported cooperative work.



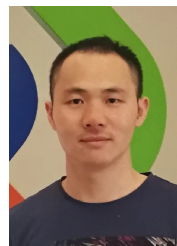
Minghui Zhou received the BS, MS, and PhD degrees in computer science from National University of Defense Technology in 1995, 1999, and 2002, respectively. She is a professor in Department of Computer Science, Peking University. She is interested in software digital sociology, i.e., understanding the relationships among people, project culture, and software product through mining the repositories of software projects. She is a member of the ACM.



Tao Wang is an assistant professor in the College of Computer at National University of Defense Technology (NUDT). He received his Ph.D. degree in Computer Science from NUDT in 2015. His work interests include open source software engineering, machine learning, data mining, and knowledge discovering in open source software.



Gang Yin is an associate professor in the College of Computer at National University of Defense Technology (NUDT). He received his Ph.D. degree in Computer Science from NUDT in 2006. He has worked in several grand research projects including National 973, 863 projects. He has published more than 60 research papers in international conferences and journals. His current research interests include distributed computing, information security, software engineering, and machine learning.



Long Lan Long Lan received the Ph.D. degree in computer science from National University of Defense Technology (NUDT) in 2017. He was a Visiting Ph.D. Student with the University of Technology, Sydney, from 2015 to 2017. He is currently a Lecturer with the College of Computer, NUDT. His research interests focus on the theory and application of artificial intelligence.



Huaimin Wang received his PH.D. in Computer Science from National University of Defense Technology (NUDT) in 1992. He is now a professor and vice-president for academic affairs of NUDT. He has been awarded the “Chang Jiang Scholars Program” professor and the Distinct Young Scholar, etc. He has published more than 100 research papers in peer-reviewed international conferences and journals. His current research interests include middleware, software agent, and trustworthy computing.