Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective

Georgios Gousios[*], Andy Zaidman[†], Margaret-Anne Storey[‡], Arie van Deursen[†]
[*] Radboud University Nijmegen, the Netherlands
Email: g.gousios@cs.ru.nl
[†] Delft University of Technology, the Netherlands
Email: {a.e.zaidman, arie.vandeursen}@tudelft.nl
[‡] University of Victoria, BC, Canada
Email: mstorey@uvic.ca

***Abstract*—In the pull-based development model, the integrator has the crucial role of managing and integrating contributions. This work focuses on the role of the integrator and investigates working habits and challenges alike. We set up an exploratory qualitative study involving a large-scale survey of 749 integrators, to which we add quantitative data from the integrator's project. Our results provide insights into the factors they consider in their decision making process to accept or reject a contribution. Our key findings are that integrators struggle to maintain the quality of their projects and have difficulties with prioritizing contributions that are to be merged. Our insights have implications for practitioners who wish to use or improve their pull-based development process, as well as for researchers striving to understand the theoretical implications of the pull-based model in software development.**

I. Introduction

Pull-based development as a distributed development model is a distinct way of collaborating in software development. In this model, the project's main repository is not shared among potential contributors; instead, contributors fork (clone) the repository and make their changes independent of each other. When a set of changes is ready to be submitted to the main repository, they create a pull request, which specifies a local branch to be merged with a branch in the main repository. A member of the project's core team (from hereon, the *integrator*[1]) is responsible to inspect the changes and integrate them into the project's main development line.

The role of the integrator is crucial. The integrator must act as a guardian for the project's quality while at the same time keeping several (often, more than ten) contributions "in-flight" through communicating modification requirements to the original contributors. Being a part of a development team, the integrator must facilitate consensus-reaching discussions and timely evaluation of the contributions. In Open Source Software (OSS) projects, the integrator is additionally taxed with enforcing an online discussion etiquette and ensuring the project's longevity by on-boarding new contributors.

The pull-based development process is quickly becoming a widely used model for distributed software development [1]. On GitHub alone, it is currently being used exclusively or complementary to the shared repository model in almost half of the collaborative projects. With GitHub hosting more than 1 million collaborative projects and competing services, such as BitBucket and Gitorious, offering similar implementations of the pull-based model, we expect the pull-based development model to become the default model for distributed software development in the years to come.

[1]Also referred to as "integration manager": http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows. We use the term integrator for brevity.

By better understanding the work practices and the challenges that integrators face while working in pull-based settings, we can inform the design of better tools to support their work and come up with best practices to facilitate efficient collaboration. To do so, we set up an exploratory qualitative investigation and survey integrators on how they use the pull-based development model in their projects. Our field of study is GitHub; using our GHTorrent database [2], we aimed our survey at integrators from high profile and high volume projects. An explicit goal is to learn from many projects rather than study a few projects in depth. We therefore use surveys as our main research instrument, generously sprinkled with open-ended questions. We motivate our survey questions based on a rigorous analysis of the existing literature and our own experience with working with and analysing the pull-based model during the last 2 years. We conducted a two-round (pilot and main) survey with 21 and 749 respondents respectively.

Our main findings reveal that integrators successfully use pull requests to solicit external contributions and we provide insights into the decision making process that integrators go through while evaluating contributions. The two key factors that integrators are concerned with in their day-to-day work are *quality* and *prioritization*. The quality phenomenon manifests itself by the explicit request of integrators that pull requests undergo code review, their concern for quality at the source code level and the presence of tests. Prioritization is also a concern for integrators as they typically need to manage large amounts of contribution requests simultaneously.

II. Background and Related Work

The goal of distributed software development methods is to allow developers to work on the same software product while being geographically and timezone dispersed [3]. The proliferation of distributed software development techniques was facilitated by the introduction of online collaboration tools such as source code version control systems and bug databases [4], [5]. The main differentiation across distributed software development methods is the process of integrating an incoming set of changes into a project's code base. This change integration process has gone through many phases, as the collaboration tools matured and adapted to changing development needs; pull-based development [1] is the latest of those developments.

In distributed software development, the first step towards integrating changes is evaluating the proposed contributions. This is a complex process, involving both technical [6], [7], [8] and social aspects [9], [10], [11].

Mockus et al. [6] analyzed two early OSS communities, Mozilla and Apache, and identified common patterns in evaluating contributions, namely the commit-then-review process. As an alternative, the Apache community also featured a review process through mailing list patch submissions. Rigby and Storey examined the peer review process in OSS mailing lists [7] and found that developers filter emails to reduce evaluation load, prioritize using progressive detail within emails containing patches and delegate by appending names to the patch email recipients. Jiang et al. [8] analyzed patch submission and acceptance in the Linux kernel project, which is using a preliminary pull-based development model, and found that, through time, contributions are becoming more frequent, while code reviews are taking less time.

As the change submission and integration models evolve, so do the evaluation processes. Bacchelli and Bird [12] refer to lightweight, branch-based peer reviews as "modern" code review. This kind of peer review is similar to the reviews taking place in pull-based development in many aspects, with an important difference: the process for accepting a contribution is pre-determined and requires sign-off by a specific number of integrators. They find that while the stated purpose of modern code review is finding defects, in practice, the benefits in knowledge transfer and team awareness outweigh those stemming from defect finding. In a similar quantitative study [13], Rigby and Bird analyzed branch-based code review processes in OSS and commercial systems and found that reviewed patches are generally very small while two reviewers find an optimal number of defects.

In recent work, Gousios et al. [1] and Tsay et al. [14] investigated quantitatively what factors underline the acceptance of contributions in pull-based development; both find similar effects, but the dominating factors (hotness of project area and social distance, respectively) are vastly different. This difference suggests that there may be no underlying processes for contribution evaluation in pull-based development that are in effect across projects. In turn, this calls for a more in-depth, qualitative study to help us understand how integrators evaluate contributions. Initial qualitative evidence on how integrators assess contributions has been reported by Pham et al. [15], but the focus of this work was the evaluation of testing practices rather than the pull-based development model.

A number of social aspects also affect the evaluation of contributions. Duchneaut found that developers looking to get their contributions accepted must become known to the core team [10]. Then, core team members would use the developer's previous actions as one of the signals for judging contributions. Similarly, Krogh et al. [9] found that projects have established implicit "joining scripts" to permit new developers to contribute to the project, according to which they examine the developers' past actions to permit access to the main repository. There is no empirical evidence on whether the developer's previous actions play a significant role in contribution assessment in the context of pull-based development; in fact, quantitative data from Gousios et al. [1] suggest otherwise. Finally, Marlow et al. [11] found that developers on GitHub use social signals, such as the developer's coding activity and the developer's social actions (e.g. following other developers), in order to form an impression of the quality of incoming contributions.

III. Research Questions

Our examination of the literature revealed that while several researchers have examined how developers evaluate contributions and collaborate in the context of OSS or, more recently, GitHub, no work has examined yet how integrators perceive pull-based development. With pull-based development rapidly rising in popularity, it is important to expand our understanding of how it works in practice and what challenges developers in general and integrators in particular face when applying it. Consequently, our first question explores how integrators employ the pull-based development model in their projects at the project level:

RQ1: *How do integrators use pull-based development in their projects?* To make the analysis easier, we further refine RQ1 in the following subquestions:

- RQ1.1 How do integrators conduct code reviews?
- RQ1.2 How do integrators merge contributions?

After a contribution has been received, the integrators must decide whether it is suitable for the project or not. Recent quantitative work identified that, across projects, simple factors such as the recent activity of the project area affected by the contribution [1] and social distance between the contributor and the integrator [14] can be used to predict whether a contribution will be accepted or not. What criteria do the integrators use to make this decision? This motivates our second research question:

RQ2: *How do integrators decide whether to accept a contribution?*

When evaluating contributions in collaborative environments, a common theme is quality assessment [6], [7], [12]. In the context of pull-based development, the asynchrony of the medium combined with its high velocity may pose additional (e.g. timing) requirements. It is beneficial to know what factors the integrators examine when evaluating the quality of a contribution and what tools they use to automate the inspection, as the results may be used to design tools that automate or centralize the evaluation process. Therefore, our third research question is as follows:

RQ3: *How do the integrators evaluate the quality of contributions?*

On busy projects, or in projects with busy integrators, contributions can pile up. It is not uncommon for large projects (for example Ruby on Rails) to have more than 100 pull requests open at any time. How do integrators cope with such a situation? How do they select the next contribution to work on when many need their immediate attention? This leads to our fourth research question:

RQ4: *How do the integrators prioritize the application of contributions?*

The challenges of online collaboration have been a very active field of study, also in the field of distributed software development [4]. The pull-based development setting is unique: the asynchrony between the production of the code and its integration in a project's code base along with the increased transparency afforded by platforms like GitHub, theoretically allow contributors and integrators to co-ordinate more efficiently. But is this so? How do integrators perceive the theoretical advantages of pull-based development in practice? By understanding the challenges that integrators face when applying pull-based development in their projects, we may better understand the limits of the pull-based method and inform the design of tools to help integrators cope with them. This leads to our final research question:

RQ5: *What key challenges do integrators face when working with the pull-based development model?*

IV. Study Design

We conducted a mixed-methods exploratory study, using mostly qualitative but also quantitative data, that consisted of two rounds of data collection. In the first round, we run a pilot survey among a limited set of selected integrators. After analyzing the results of the first round, we identified emerging themes (specifically, quality and prioritization), which we address by including related questions in the second round. The survey results of the second round were further augmented, and partitioned by, quantitative results for each specific project. In this section, we describe our research method in detail.

A. Protocol

Since our aim is to learn from a large number of projects, we used surveys which scale well.

Survey Design The study took place in two rounds, a pilot round that gave us the opportunity to field test our initial questions and the final round through which we gathered the actual responses.

Both surveys were split into three logical sections; demographic information, multiple choice or Likert-scale questions and open-ended questions. The open-ended questions were intermixed with multiple choice ones; usually, the developer had to answer an open-ended question and then a related one with fixed answers. To further elicit the developers' opinions, in all questions that had predefined answers but no related open-ended question, we included an optional "Other" response. Finally, we intentionally used even Likert scales to force participants to make a choice. Overall, and excluding demographic questions, the survey included 7 open-ended questions, 7 Likert scale questions with an optional open-ended response and 6 multiple choice questions with no optional fields. The survey could be filled in in about 15 minutes.

The purpose of the survey pilot was to identify themes on which we should focus the main survey. As such, the pilot survey included fewer open-ended questions, but all multiple choice questions have optional open-ended reply fields. This allowed us to test our initial question set for strongly correlated answers (we removed several potential answers from multiple choice questions) and identified two topics, namely quality and prioritization which we addressed in the main survey round.

Attracting participants In previous work [1], we presented evidence that most repositories on GitHub are inactive, single user projects. To ensure that our sample consisted of repositories that make effective and large scale use of pull requests, we selected all repositories in our GHTorrent dataset [2] that have received at least one pull request for each week in the year 2013 (3,400 repositories). For the selected repositories, we extracted the top pull request integrators, as identified by the number of pull requests that they have merged, and build our correspondence list.

For the pilot phase, we emailed 250 of those integrators randomly and received 21 answers (8% answer rate). For the data collection phase, we emailed integrators from the remaining 3,150 projects and received 749 answers (23% answer rate). The survey's web address was sent by personal email to all participants. We did not restrict access to the survey to invited users only. In fact, several survey respondents forwarded the survey to colleagues or advertised it on social media (Twitter) without our consent. After comparing the response set with the original set of projects we contacted, we found that 35% of the responses came through third party advertising of the survey. The survey ran from April 14 to May 1, 2014.

To encourage participation, we created a customized project report for each project in our correspondence list. The report included plots on the project's performance in handling pull requests (e.g. mean close time) on a monthly basis. The reports for all projects have been published online[2] and since then have been widely circulated among developers. Of the 749 survey respondents, 138 also expressed gratitude for their report through email.

B. Participants

The majority of our respondents self-identified as project owners (71%), while 57% work for industry. Most of them also have more than 7 years of software development experience (81%) and considerable experience (> 3 years) in geographically distributed software development (76%).

[2]http://ghtorrent.org/pullreq-perf/

To identify the leading groups of respondents based on the combined effect of experience, role in the project and work place, we ran the kmodes clustering algorithm (a variation of kmeans for categorical data) on the dataset. The clustering results revealed that $\frac{1}{3}$ of the respondents (275/749) are project owners with more that 7 years of industrial experience; of those, around 40% (108/275) also worked exclusively on the projects they responded about.

C. Analysis

We applied manual coding on the seven open-ended questions as follows: initially, three of the four authors individually coded a different set of 50 (out of 750) answers for each question. At least one and up to three codes were applied to each answer. The order of code application reflected the emphasis each answer gave on the code topic. The extracted codes were then grouped together and processed to remove duplicates and, in cases, to generalize or specialize them. The new codes were then applied on all answers by the first author. When new codes emerged, they were integrated in the code set. On average, 30% more codes were discovered because we decided to code the full dataset.

In the survey, we asked integrators to optionally report a single repository name for which they handle most pull requests. 88% of the respondents did so. For the remaining 83 answers, we either resolved the repository names from the developer's emails (since integrators were invited to participate based on a specific email), or selected the most active project the developer managed pull requests for, while we also fixed typos in repository names. We excluded from further analysis answers for which we could not obtain a repository name (61 answers). After we resolved the repository names, we augmented the survey dataset with information from the GHTorrent database [2]. Specifically, for each project, we calculated the mean number of pull requests per month and the mean number of integrators for the period July 2013 to July 2014. Using those metrics, and for each one of them, we split the project population in three equally sized groups (small, medium and large). Finally, we excluded answers from projects that received no pull request in this time frame (14 answers). None of these were in our original contact list.

V. Results

In this section, we present our findings per research question. To enable traceability, we include direct quotes from integrators along with the answer identified in our dataset (e.g. R1 corresponds to answer 1). Similarly, in the case of coded open-ended questions, we present the discovered codes *slanted.*

A. RQ1: How do integrators use pull-based development in their projects?

1) Overall use: To understand why and how projects use the pull-based development model, we asked integrators a multiple choice question that included the union of potential uses of pull requests that have been reported in the literature [1], [16], [15]. Respondents also had the opportunity to report other uses not in our list.

Overwhelmingly, 80% of the integrators use the pull-based development model for doing code reviews and 80% to resolve issues. Perhaps more interesting is that half of the integrators use pull requests to discuss new features (as R710 commented: "*experimenting with changes to get a feel if you are on the right path*"). This is a variation of the GitHub-promoted way of working with pull requests,[3] where a pull request is opened as early as possible to invite discussion on the developed feature.

60% of the integrators use pull requests to solicit contributions from the community (people with no direct commit access to the repository), which seems low given the open nature of the GitHub platform. We examined this response quantitatively, using the GHTorrent database: indeed for 39% percent of the projects that responded, no pull request originated from the project community. There is a small overlap (30%) between projects responding that they do not use pull requests to solicit contributions from the community and those that actually did not receive a pull request. Moreover, another 28% of the projects reported that they have used pull requests to solicit contributions from the community even though they did not receive any external pull requests.

Only 4% (or 29) of the respondents indicated that they use pull requests for something else. The analysis of the answers reveals that the majority of the replies nevertheless aligns with the offered choice answers with two notable exceptions. Respondent R635 mentions that they use pull requests in "*every commit we make. We have a policy of having every commit, even bumping up version number for next release, coming in on a PR.*". The project has effectively turned pull requests into a meta-version control system, one that only allows reviewed code to be merged. This merging behaviour is also in place within Microsoft [12] and in the Android project [13]. Another integrator is using pull requests as a time machine mechanism: R521: "*Ideally, any change, because using PRs makes it easier to rollback a change if needed*".

2) Code reviews: In the time between a pull request submission and before it is accepted, it becomes a subject of inspection. 75% of the projects indicate that they do explicit code reviews on all contributions (only 7% of the projects do not review their pull requests using GitHub, but those have specified alternative ways of doing code reviews as described below). On GitHub, anyone can participate in the inspection process. 50% of the integrators report that the project's community actively participates in code reviews; this is in contrast with Gousios et al. [1], where we found that in all projects we examined, the community discussing pull requests was bigger than the core team.

In current code reviewing practices, using tools such as Gerrit [13] or Codeflow [12], code review comments are intermingled with code and a predetermined approval process is in place. GitHub offers a more liberal code reviewing system where users can provide comments on either the pull

[3]https://github.com/blog/1124 Accessed Jul 2014

request as a whole, the pull request code or even in individual commits comprising the pull request, but imposes no approval process. 75% of the integrators use inline code comments in the pull request to do code reviews; only 8% of the integrators report that they use commit comments. The absence of strict acceptance process support has created a market for code reviewing tools: of the 7% (or 52) of the integrators that indicated they are doing code reviews in another way, 20% (or 10) mentioned that they are explicitly using a different tool for doing code reviews.

Projects have established processes for doing code reviews. One of them is delegation; 42% of the integrators delegate a code review if they are not familiar with the code under review. Delegation is again not a strictly defined process on GitHub; by convention, it can occur by referencing (@username) a user name in the pull request body, but integrators report other ways to delegate work: for example, R62 uses video conferencing to discuss pull requests and assign work load, while others (e.g. R577, R587) use external tools with support for delegation. Another process is implicit sign-off: at least 20 integrators reported that multiple developers are required to review a pull request to ensure high quality. Typically this is 2 reviewers, e.g. R481: "*We have a rule that at least 2 of the core developers must review the code on all pull requests.*". Rigby and Bird also report a similar finding in Gerrit-based industrial projects [13].

3) Integrating Changes: When the inspection process finishes and the contributions are deemed satisfactory, they can be merged. A pull request can only be merged by core team members. The versatility of Git enables pull requests to be merged in various ways, with different levels of preservation of the original source code properties. Briefly, a pull request can be integrated either through GitHub's facilities or a combination of low level git commands, such as merge or cherry-pick.

We gave integrators a list of 4 ways to perform merges, as identified in [17], and asked them how often they use them, but also allowed them to describe their own. In 79% of the cases, integrators use the GitHub web interface "often or always" to do a merge; this number is actually close to what we obtained by quantitatively analyzing pull requests in [17] and [1]. Only in 8% and 1% of the cases do integrators resort to cherry-picking or textual patches respectively to do the merge.

As identified by the integrators in the comments, the command-line git tool is mostly used in advanced merging scenarios where conflicts might occur. 4% (or 28) of the respondents mentioned that they are using rebasing (history rewriting) in the following ways: i) placing the new commits in the source branch on top of the current ones in the target branch (e.g. R306 and R316), which effectively merges the two branches while avoiding redundant merge commits, and ii) asking the contributor to squash pull request commits into one before submitting the pull request. Moreover, integrators indicated that they allow their continuous integration system to do the merge (e.g. R157) or use scripts to automate merges between feature branches (e.g. R321).

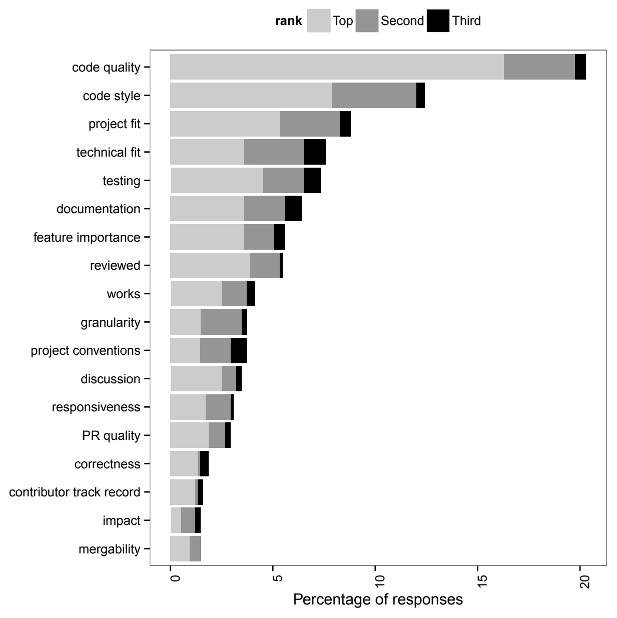

Fig. 1: Signals used by integrators when deciding on whether a contribution will be accepted or not.

Overall, integrators emphasize the preservation of commit metadata by avoiding textual patches and cherry-picking, while some of them use history rewriting to avoid the formation of complicated networks of branches and merges.

RQ1: *Integrators successfully use the pull-based model to accommodate code reviews, discuss new features and solicit external contributions. 75% of the integrators conduct explicit code reviews on all contributions. Integrators prefer merges that preserve commit metadata.*

B. RQ2: How do integrators decide whether to accept a contribution

The second research question elicits the signals that integrators use to decide on the fate of a contribution. We asked integrators an optional open-ended question and received 324 answers. The results are summarized in Figure 1.

The most important factor leading to acceptance of a contribution is its quality. Quality has many manifestations in our response set; integrators examine the *source code quality* and *code style* of incoming code, along with its *documentation* and *granularity*: "*Code style and whether or not it matches project style. Overall programming practice, lack of hacks and workarounds.*" (R32). At a higher level, they also examine the quality of the commit set and whether it adheres to the *project conventions* for submitting pull requests.

A second signal that the integrators examine is *project fit*. As respondent R229 states: "*The most important factor is if the proposed pull request is in line with the goals and target of the project*". A variation is *technical fit*: does the code fit the technical design of the project (R90: "*Most important to us is that the contribution is in keeping with the spirit of the project's other APIs, and that its newly introduced code follow the total and functional style of the rest of the codebase*"). Integrators also examine the *importance of the fix/feature* with respect to the current priorities of the project. This is common in case of bug fixes: "*If it fixes a serious bug with minimal changes, it's more likely to be accepted.*" (R131).

A third theme that emerged from the integrator responses is testing. Apart from assessing the quality of contributions using higher level signals, integrators also need to assess whether the contributed code actually works. Initially, integrators treat the *existence of testing code* in the pull request as a positive signal. Success of test runs by a continuous integration system also reinforces trust in the code: "*All tests must pass integration testing on all supported platforms...*"(R94). Finally, integrators resort to *manual testing* if automated testing does does not allow them to build enough confidence: "*If other developers verified the changes in their own clones and all went fine, then we accept.*" (R156).

It is interesting to note that the track record of the contributors is ranked low in the integrator check list. This is in line with our earlier analysis of pull requests, in which we did not see a difference in treatment of pull requests from the core team or from the project's community [1].

Finally, technical factors such as whether the contribution is in a mergeable state, its impact on the source code or its correctness are not very important for the eventual decision to merge to the majority of respondents. In such cases, integrators can simply postpone decisions until fixes are being provided by the contributors: "*...occasionally I go through discussion with committer on how to do things better or keep the code-style held in the whole project*" (R300). The postponing effect has also been observed by Rigby and Storey [7].

RQ2: *Integrators decide to accept a contribution based on its quality and its degree of fit to the project's roadmap and technical design.*

C. RQ3: What factors do the integrators use to examine the quality of contributions?

When examining contributions, quality is among the top priorities for developers. With this research question, we explore how integrators perceive quality and what tools they use to assess it, by means of a pair of compulsory open-ended and multiple choice questions. The results are summarized in Figure 2.

1) Perception: One of the top priorities for integrators when evaluating pull request quality is *conformance*. Conformance can have multiple readings: For R39, conformance means "*it matches the project's current style (or at least improve upon it)*" (*project style*) while for R155 conformance is to be evaluated against fitting with internal API usage rules (*architecture fit*). Many integrators also examine conformance against the programming language's style idioms (e.g. PEP8 for Python code). Integrators expect the contributed code to cause minor friction with their existing code base and they try to minimize it by enforcing rules on what they accept.

Integrators often relate contribution quality to the quality of the source code it contains. To evaluate source code quality, they mostly examine non-functional characteristics of the changes. Source code that is *understandable* and *elegant*, has

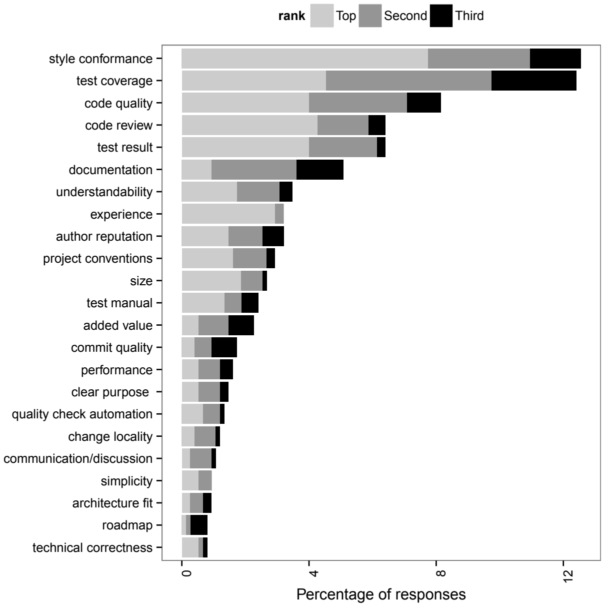

Fig. 2: Factors that integrators examine when evaluating the quality of contributions

good *documentation* and provides clear *added value* to the project with minimal *impact* is preferred.

Apart from source code, the integrators use characteristics of the pull request as proxies to evaluate the quality of the submission. The quality (or even the existence) of the pull request *documentation* signifies an increased attention to detail by the submitter: "*A submitter who includes a clear description of what their pull request does have usually put more time and thought into their submission*" (R605). The integrators also examine the *commit organization* in the pull request: "*well written commit messages; one commit about a single subsystem — each commit compiles separately*" (R610) and its *size*. In the latter case, the integrators value small pull requests as it is easier to assess their impact (R246: "*...the code has the minimum number of lines needed to do what it's supposed to do*" or R330: "*is the diff minimal?*").

Testing plays an important role in evaluating submissions. Initially, the very existence of tests in the pull request is perceived as a positive signal. The integrators also examine whether the changes in the pull request are covered by existing or new tests (*test coverage*), while, in 4% of the cases, they report that they exercise the changes manually (*manual testing*). Moreover, in performance-critical code, *performance* degradation is frowned upon and in some cases, integrators require proof that performance is not affected by the proposed change, e.g. in R72: "*Performance related changes require test data or a test case*".

Finally, integrators use social signals to build trust for the examined contribution. The most important one is the *contributor's reputation*. The integrators build a mental profile for the contributor by evaluating their track record within the project (R405: "*Who submitted the PR and what history did we have with him/her?*") or by searching information about the contributor's work in other projects (R445: "*looking at* *the other contributions in other projects of the pull author*"). Some integrators also use *interpersonal relationships* to make judgements for the contributor and, by proxy, for their work. The process of impression building through social signals has been further elaborated by Marlow et al. [11].

2) Tools: Quality evaluations can be supported by tools. To evaluate how often projects use tools, we gave integrators a selection of tools and asked them which ones they use in their projects. The vast majority (75%) of projects use *continuous integration*, either in hosted services or in standalone setups. Continuous integration services, such as Travis and CloudBees, allow projects to run their test suites against incoming pull requests, while integration with GitHub enables them to update pull requests with test outcomes. On the other hand, few projects use more dedicated software quality tools such as *metric calculators* (15%) or *coverage reports* (18%). It is interesting to note that practically all (98%) projects that use more advanced quality tools, run them through continuous integration.

99 integrators responded that they are using other tools. By going through the responses, we see that integrators use a rather limited toolset. Specifically, only a handful of integrators reported that they are using *linting* tools[4] while dedicated *static analysis* tools are used in just two large scale C++ projects in our sample. In two more cases, the integrators reported that they rely on the language's *type system* to eliminate bugs. Finally, the majority of integrators answered that they evaluate the quality manually (e.g. R291: "*my brain is a powerful testing environment*" or R353: "*good eyes and many eyes*") even when they were asked what tools they are using to do so.

RQ3: *Top priorities for integrators when evaluating contribution quality include conformance to project style and architecture, source code quality and test coverage. Integrators use few quality evaluation tools other than continuous integration.*

D. RQ4: How do the integrators prioritize the application of contributions?

Our fourth research question examines the factors integrators use to prioritize their work on evaluating contributions. To discover them, we asked integrators a compulsory open-ended question. The results are summarized in Figure 3.

The first thing that integrators examine is the contribution's *urgency*. In case of bug-fixing contributions, the *criticality of the fix* is the most important feature to prioritize by. Integrators examine at least the following factors to assess criticality: i) the contribution fixes a security issue, ii) the contribution fixes a serious new bug, iii) the contribution fixes a bug that other projects depend upon, and iv) number of issues blocked by the unsolved bug.

In the case of a contribution implementing new features, integrators examine whether the contribution implements customer requested features or features required for the devel-

[4]Tools that find common mistakes, e.g. uninitialized variables

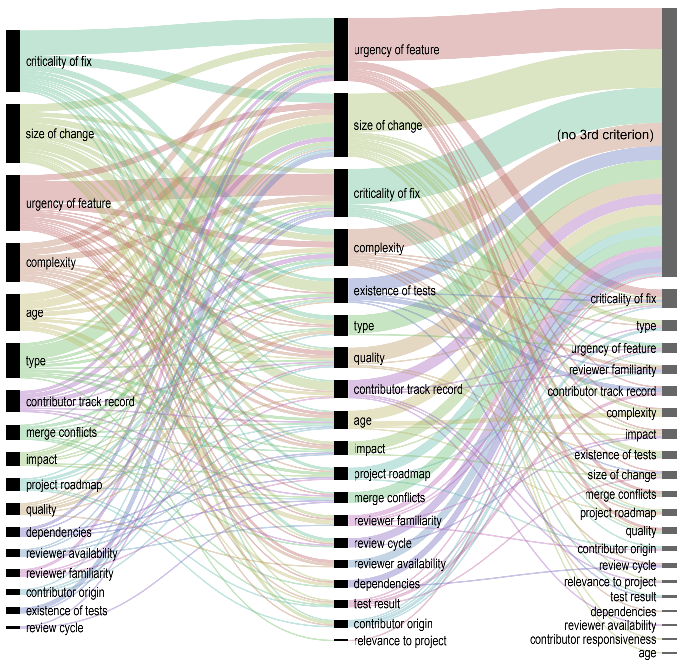

Fig. 3: Factors used for prioritization and their order of application (left to right) as reported by integrators. The thickness of each line corresponds to the frequency the particular prioritization order appeared in our response set.

opment of other features. Several integrators also mentioned that they just examine the *type* of the contribution before its criticality; it is usually project policy to handle bug fixing contributions before enhancements, as is the case with R446: "*Bug fixes first, then new features. Only if all bug fix pull requests are treated.*"

The pull request *age* plays an important role in prioritization for integrators. It is interesting to note that many integrators prefer a first-in, first-out treatment of the pull requests before applying other prioritization criteria. Similarly, easy to assess (and therefore less *complex*) pull requests are preferred by integrators. The *size* of the patch, even through usually related to complexity, is used to quickly filter out small, easy to integrate contribution and process them first (e.g. R490: "*The lower the number of lines/files changes, the more likely I am to process it first.*")

The *contributor's track record* is a relatively important factor for prioritization and usually known contributors get higher priority. As R82 states it: "*If I know the person, they get high priority. Sorry, strangers.*". A related criterion is the *contributor's origin*; if the contributor is another core team member or, in business settings, a colleague, some projects assign priorities to his/her contributions (e.g. R106, R183, R411), while some others specifically favour community contributions (e.g. R161, R398).

Finally, it is interesting to note that 18% of all integrators in our sample are not using any prioritization processes at all.

When prioritizing contributions, integrators must apply multiple criteria in a specific sequence. Figure 3 depicts the frequencies of prioritization criteria usage for all reported application sequences. What we can see is that criticality, urgency and change size contribute to most prioritization criteria application sequences, while most integrators report that they apply at most two prioritization criteria.

> ***RQ4:*** *Integrators prioritize contributions by examining their criticality (in case of bug fixes), their urgency (in case of new features) and their size. Bug fixes are commonly given higher priority. One fifth of the integrators do not prioritize.*

E. RQ5: What key challenges do integrators face when working with the pull-based development model?

We asked integrators an optional open-ended question and received 410 answers. We found two broad categories of challenges: *technical* challenges hamper the integrator's ability to work effectively, while *social* challenges make it difficult for integrators to work efficiently with other project members.

1) Technical challenges: At the project level, *maintaining quality* is what most integrators perceive as a serious challenge. As incoming code contributions mostly originate from non-trusted sources, adequate reviewing may be required by integrators familiar with the project area affected by it. *Reviewer availability* is not guaranteed, especially in projects with no funded developers. Often, integrators have to deal with solutions tuned to a particular contributor requirement or an edge case; asking the contributor to *generalize* them to fit the project goals is not straightforward. A related issue is *feature isolation*; contributors submit pull requests that contain multiple features and affect multiple areas of the project. As put by R509: "*Huge, unwieldy, complected bundles of 'hey I added a LOT of features and fixes ALL AT ONCE!' that are hell to review and that I'd like to *partially* reject if only the parts were in any way separable...*".

Several issues are aggravated the bigger or more popular a project is. Integrators of popular projects mentioned that the *volume* of incoming contributions is just too big (e.g. Ruby on Rails receives on average 7 new pull requests per day) consequently, they see triaging and work prioritization as challenges. As requests are kept on the project queue, they *age*: the project moves ahead in terms of functionality or architecture and then it is difficult to merge them without (real or logical) *conflicts*. Moreover, it is not straightforward to assess the *impact* of stale pull requests on the current state of the project or on each other.

Another category of technical challenges is related to the experience of the contributor. Integrators note that aspiring contributors often ignore the *project processes* for submitting pull requests leading to unnecessary communication rounds. When less experienced developers or regular users attempt to submit a pull request, they often lack basic *git skills* (e.g. R42: "*Lack of knowledge of git from contributors; most don't know how to resolve a merge conflict.*"). New contributors can be a valuable resource for a project; integrators report that they avoid confrontation in an effort to onboard new users.

Many of the challenges reported by the integrators are bound to the distributed nature of pull-based development. Lack of *responsiveness* on behalf of the contributor hurts the

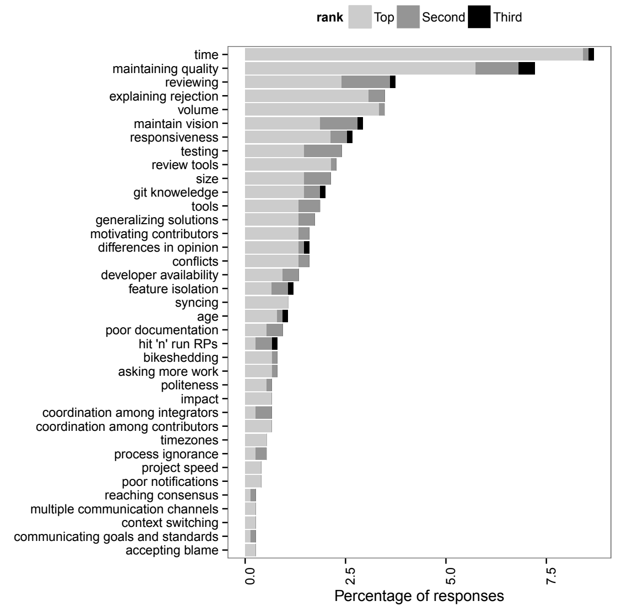

Fig. 4: Biggest challenges when working with the pull-based development model.

code review process and, by extension, project flow. This is especially pronounced in the case of *hit and run* pull requests,[5] as they place additional reviewing and implementation burden on the integrator team. Integrators mention that the lack of centralized co-ordination with respect to project goals can lead to "*chaos. Lots of people trying to reach the same goal without coordinating*" (R155).

Finally, integrators also report inefficiencies in the GitHub platform itself. Specifically, many integrators complained about the quality of the *code review* tool offered by GitHub (R567: "*A good code review tool with code analysis possibilities can help*") and made comparisons to their favourite ones (e.g. R288: "*The mechanism itself is a huge step backwards from Reviewboard*") while others did not like the way GitHub handles notifications (e.g. R514: "*Sifting through the GitHub information flood to find what, if any, I should address.*").

2) Social challenges: Integrators often have to make decisions that affect the social dynamics of the project. Integrators reported that *explaining the reasons for rejection* is one of the most challenging parts of their job as *hurting* the contributor's *feelings* is something they seek to avoid. As R255 explains: "*Telling people that something is wrong without hurting their feelings or giving them an incorrect idea of my intentions.*". Similarly, integrators find that *asking for more work* from the contributors (e.g. as a result of a code review) can be difficult at times, as they "*... worry about alienating our valued contributors*" (R635). *Motivating contributors* to keep working on the project, even in the face of rejected contributions, is not easy for integrators either.

[5] R708 describes hit and run pull requests nicely: "*They (contributors) send a pull request with a bug but when I ask them to fix them then they just vanish and don't respond to GitHub e-mails.*"

Reaching consensus through the pull request comment mechanism can be challenging. Integrators often find themselves involved in a balancing act of trying to *maintain* their own *vision* of the project's future and incorporating (or rejecting) contributions that are tuned to the contributor's needs. *Differences in opinion* compared to the relative anonymity of the pull request comment mechanism can lead to unpleasant situations. Integrators may need to take action to maintain discussion etiquette (e.g. R449 "*Dealing with loud and trigger-happy developers.*"), enforce *politeness* rules or to stop long, unhelpful (*bikeshedding*) discussions (R586: "*be objective and avoid off-topics in discussions*"). *Multiple communication channels* are not helping either; integrators find it difficult to synchronize between multiple sources.

On a more personal level, integrators find it difficult to handle the workload imposed by the open submission process afforded by the pull-based development model. For many of our respondents, managing contributions is not their main job; consequently finding *free time* to devote on handling a pull request and *context switching* between various tasks puts a burden on integrators. As R470 notes: "*Managing pull requests is not my full-time job, but it is a component of it. Mostly it is difficult to keep track of them while also completing my other tasks.*".

RQ5: *Integrators are struggling to maintain quality and mention feature isolation and total volume as key technical challenges. Social challenges include motivating contributors to keep working on the project, reaching consensus through the pull request mechanism and explaining reasons for rejection without discouraging contributors.*

VI. Discussion

In this section, we compare and contrast our findings with existing work and present future work directions.

A. Quality

Throughout our analysis, the issue of quality evaluation was recurring. The respondents directly linked quality with acceptance while also described maintaining quality as a big challenge. According to integrators, quality emerges from attention to detail; code style, documentation, commit formatting and adherence to project conventions all help to build confidence in the contribution. The issue of quality evaluation has been repeatedly mentioned in works on patch submission [7], [18], lightweight code review [12], [13] and testing [15]; in this sense, our work reinforces earlier findings. In addition, we document in detail what factors integrators examine in contributions when doing quality assessments.

An open question is how to efficiently automate the quality evaluation for pull requests. While tools that automate the evaluation of many tasks that the developers do to determine quality (e.g. code style analyzers, test coverage, metrics for software quality, impact analysis etc) do exist, we have seen that developers go little beyond testing and continuous integration. To solve this issue, one could envisage a pluggable platform that, given a pull request update, runs a suite of tools and automatically updates the pull request with a configurable quality score. For the platform to be useful, it will have to automatically learn from and adapt to project-specific behaviours.

B. Testing

Integrators overwhelmingly use testing as a safety net when examining contributions. The inclusion of tests in a contribution is perceived as a positive signal, while (reverse) coverage is evaluated by many integrators. 75% of our respondents run tests automatically through continuous integration services. Pham et al. examined how testing works on GitHub [15]; our work confirms many of their findings (e.g. use of testing as a quality signal, manual examination when continuous integration fails) and complements it with more quantitative data about test diffusion on GitHub projects. Moreover, it is interesting to pinpoint the contradiction with the results of our previous work [1], where we found that inclusion of test code in a contribution was not a strong factor influencing either the decision to accept or the time to decide (Tsay et al. [14] report a similar result). We speculate that this difference is due to how we modeled test inclusion (continuous rather than a dichotomous feature) in our previous study.

C. Work Prioritization

In large projects, integrators cannot keep up with the volume of incoming contributions. A potential solution could be a recommendation system that provides hints on which contributions need the integrator's immediate attention. Existing work on assisted bug triaging (e.g. [19] or [20]) is not directly applicable to the pull-based model, as a pull request is not necessarily as static as a bug report. Researchers might need to come up with different methods of work prioritization that take into account the liveness and asynchrony of the pull-request model. Our analysis of how developers prioritize contribution is a first step in this direction.

D. Developer Track Records

One finding of this work is that a developer's track record, while present in our response set, is not a commonly used criterion to assess or prioritize contributions by. With the raise of transparent work environments [16], and based on previous work on the subject [9], [11], one would expect that the developer's track record would be used by the majority of integrators to make inferences about the quality of incoming contributions. Despite this, the track record is mostly used as an auxiliary signal; in both Figure 2 and Figure 3, we can see that developers equally mentioned the track record as top and second criterion for quality evaluation and prioritization.

E. Community Building

Community building through collaboration has been studied extensively in the context of OSS projects [9], [21], [22]. A common theme in those studies is that recruitment of new developers can be challenging [21], as core teams are reluctant to give access to the main repository without an initiation process [9]. Integrators in our study actually mentioned the opposite: it is maintaining the community momentum and motivating contributors to do more work that is not easy. Through transparency [16] and lowered barriers to participation [15], [1], the pull-based model can act as glue for communities build around projects, if integrators are keen enough on fostering their project's communities by helping newcomers cope with tools and project processes, prioritizing the examination of community contributions and, in the extreme case, not rejecting unwanted contributions.

F. A modern theory of software change

In the recent years, we are witnessing that collaborative, lightweight code review is increasingly becoming the default mechanism for integrating changes, in both collocated [12] and distributed [13], [1] development. Effectively, the pull request (in various forms) is becoming the atomic unit of software change. Existing works (e.g. [23], [24]) neither did anticipate lightweight code reviews nor asynchronous integration of changes. This work can contribute to theory building by providing empirical evidence about the common practices of pull-based development.

VII. Limitations

We carefully designed the survey to gain insight into the work practices and challenges faced by integrators in pull-based development. We thoughtfully crafted the wording of each of the questions (to avoid ambiguous or leading questions), refining them through small pilot tests and consults with other researchers with survey research expertise, and refined the questions yet further through a larger pilot study. The response categories we supplied for many of the questions were based on the existing literature, and were likewise refined through the pilot studies. For the questions that had multiple response options, we supplied an additional "other" field which was used to uncover responses not considered that we later coded. Despite our best efforts, this work may be subject to the following limitations:

Generalizability: Since we did purposive sampling from the population of integrators, the findings may not apply to other populations of integrators (e.g. developers using other tools, integrators that work private projects on GitHub or integrators that are not in the top three integrators for a given project). Moreover, in previous work [1], we found that the median number of pull requests across repositories is 2; in our sample, the smallest project had more than 400. We expect that if the study is repeated using random sampling for projects, the results will be slightly different, as the average project does not use pull requests in a high capacity. Furthermore, the integrators that responded to our survey may have introduced an additional bias to the results (non-responders may have had different insights or opinions).

Researcher bias: It is possible that researcher bias may have influenced the wording of questions (perhaps to be leading) as well as the coding of the open ended questions. As discussed above, we tested the questions through pilots and had experts evaluate it for this concern. In terms of the analysis of the open ended questions, we conducted a pilot study, and three of us separately coded a sample of the responses to derive these codes.

Research reactivity: The ordering of questions (one may provide context for the next one), the open ended questions, as well as a respondent's possible tendency to to appear in a positive light (for example, they wish to think they are fair or logical), may have influenced the accuracy of the answers provided.

VIII. Conclusions

Our work studies the pull-based development model from the integrator's perspective. Our goal is to better understand the work practices of integrators working with the pull-based development model and to identify the challenges they face when integrating contributions. The key contributions of this paper are as follows:

- A novel way of using the GHTorrent dataset to generate targeted reports, large scale surveys and augmenting qualitative datasets with quantitative data.
- A publicly available data set with 749 anonymized survey answers.
- A thorough analysis of survey data resulting in answers to our research questions on topics such as work practices in pull-based development, quality evaluation of contributions, work prioritization and open challenges when working with pull requests.

Our anonymized response set, our coded open-ended questions and custom-built R-based analysis and plotting tools are available in the Github repository gousiosg/pullreqs-integrators. This data set complements existing quantitative data sets (e.g. our own GHTorrent data set) and provides much needed context for analyzing and interpreting that data. Furthermore, our survey brings additional insights to the insightful but smaller scale interviews that have been conducted by other researchers on the pull based model (e.g. [16], [11], [15], [14]). We welcome replications of this work; potential directions include replications with integrators that (1) use different (non-GitHub) repositories, e.g., Bitbucket, (2) work on private repositories, and (3) work on non-pull request intensive projects. These replications will help in moving towards a theory of how pull-based development impacts distributed software development.

Last but not least, our findings point to several research directions (see Section VI) and have implications for both practice and research. Based on our results, integrators can structure their contribution evaluation processes in an optimized way and be informed about common pitfalls in community management. Researchers can reuse our research methods and datasets to conduct large scale, mixed-methods research, while they can use our research findings as a basis to drive their work on pull request quality evaluation and work prioritization tools.

Acknowledgements The authors would like to thank the survey participants for their time. This work has been partially funded by the NWO 639.022.314 — TestRoots project.

References

[1] G. Gousios, M. Pinzger, and A. van Deursen, “An exploratory study of the pull-based software development model,” in *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 345–355.

[2] G. Gousios, “The GHTorrent dataset and tool suite,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236.

[3] D. Gumm, “Distribution dimensions in software development projects: A taxonomy,” *Software, IEEE*, vol. 23, no. 5, pp. 45 –51, sept.-oct. 2006.

[4] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, “Identification of coordination requirements: implications for the design of collaboration and awareness tools,” in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. New York, NY, USA: ACM, 2006, pp. 353–362.

[5] J. Whitehead, “Collaboration in software engineering: A roadmap,” in *2007 Future of Software Engineering*, ser. FOSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 214–225.

[6] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and Mozilla,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, 2002.

[7] P. C. Rigby and M.-A. Storey, “Understanding broadcast based peer review on open source software projects,” in *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 541–550.

[8] Y. Jiang, B. Adams, and D. M. German, “Will my patch make it? and how fast?: case study on the Linux kernel,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 101–110.

[9] G. von Krogh, S. Spaeth, and K. R. Lakhani, “Community, joining, and specialization in open source software innovation: a case study,” *Research Policy*, vol. 32, no. 7, pp. 1217 – 1241, 2003, open Source Software Development.

[10] N. Duchneaut, “Socialization in an open source software community: A socio-technical analysis,” *Computer Supported Cooperative Work (CSCW)*, vol. 14, no. 4, pp. 323–368, 2005.

[11] J. Marlow, L. Dabbish, and J. Herbsleb, “Impression formation in online peer production: activity traces and personal profiles in GitHub,” in *Proceedings of the 2013 conference on Computer supported cooperative work*. New York, NY, USA: ACM, 2013, pp. 117–128.

[12] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 712–721.

[13] P. C. Rigby and C. Bird, “Convergent contemporary software peer review practices,” in *Proceedings of the 2013 9th Joint Meeting on Foundations*

[13] P. C. Rigby and C. Bird, “Convergent contemporary software peer review practices,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2013, pp. 202–212.

[14] J. Tsay, L. Dabbish, and J. Herbsleb, “Influence of social and technical factors for evaluating contribution in GitHub,” in *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 356–366.

[15] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider, “Creating a shared understanding of testing culture on a social coding site,” in *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 112–121.

[16] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, “Social coding in GitHub: transparency and collaboration in an open software repository,” in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. New York, NY, USA: ACM, 2012, pp. 1277–1286.

[17] G. Gousios and A. Zaidman, “A dataset for pull-based development research,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 368–371.

[18] O. Baysal, O. Kononenko, R. Holmes, and M. Godfrey, “The secret life of patches: A Firefox case study,” in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, Oct 2012, pp. 447–455.

[19] G. Jeong, S. Kim, and T. Zimmermann, “Improving bug triage with bug tossing graphs,” in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE ’09. New York, NY, USA: ACM, 2009, pp. 111–120.

[20] P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, “Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows,” in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, May 2010, pp. 495–504.

[21] M. Stürmer and T. Myrach, “Open source community building,” *Licentiate, University of Bern*, 2005.

[22] J. West and S. O’Mahony, “Contrasting community building in sponsored and community founded open source projects,” in *HICSS’05. Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. IEEE, 2005, pp. 196c–196c.

[23] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, “Towards a taxonomy of software change,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, pp. 309–332, 2005.

[24] P. C. Rigby, D. M. German, and M.-A. Storey, “Open source software peer review practices: a case study of the Apache server,” in *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 541–550.