An Exploratory Study of the Pull-Based Software Development Model

Georgios Gousios
Delft University of Technology
Delft, The Netherlands
G.Gousios@tudelft.nl

Martin Pinzger
University of Klagenfurt
Klagenfurt, Austria
martin.pinzger@aau.at

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vandeursen@tudelft.nl

ABSTRACT

The advent of distributed version control systems has led to the development of a new paradigm for distributed software development; instead of pushing changes to a central repository, developers pull them from other repositories and merge them locally. Various code hosting sites, notably Github, have tapped on the opportunity to facilitate pull-based development by offering workflow support tools, such as code reviewing systems and integrated issue trackers. In this work, we explore how pull-based software development works, first on the GHTorrent corpus and then on a carefully selected sample of 291 projects. We find that the pull request model offers fast turnaround, increased opportunities for community engagement and decreased time to incorporate contributions. We show that a relatively small number of factors affect both the decision to merge a pull request and the time to process it. We also examine the reasons for pull request rejection and find that technical ones are only a small minority.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.9 [**Software Engineering**]: Management—*Programming teams*

General Terms

Management

Keywords

Pull-based development, pull request, distributed software development, empirical software engineering

1. INTRODUCTION

Pull-based development is an emerging paradigm for distributed software development. As more developers appreciate isolated development and branching [7], more projects, both closed source and, especially, open source, are being migrated to code hosting sites such as Github and Bitbucket, which provide support for pull-based development [2]. A unique characteristic of such sites is that they allow any user to clone any public repository. The clone creates a public project that belongs to the user that cloned it, so the user can modify the repository without being part of the development team. Furthermore, such sites automate the selective contribution of commits from the clone to the source through pull requests.

Pull requests as a distributed development model in general, and as implemented by Github in particular, form a new method for collaborating on distributed software development. The novelty lays in the decoupling of the development effort from the decision to incorporate the results of the development in the code base. By separating the concerns of building artifacts and integrating changes, work is cleanly distributed between a contributor team that submits, often occasional, changes to be considered for merging and a core team that oversees the merge process, providing feedback, conducting tests, requesting changes, and finally accepting the contributions.

Previous work has identified the processes of collaboration in distributed development through patch submission and acceptance [23, 5, 32]. There are many similarities to the way pull requests work; for example, similar work team structures emerge, since typically pull requests go through an assessment process. What pull requests offer in addition is process automation and centralization of information. With pull requests, the code does not have to leave the revision control system, and therefore it can be versioned across repositories, while authorship information is effortlessly maintained. Communication about the change is context-specific, being rooted on a single pull request. Moreover, the review mechanism that Github incorporates has the additional effect of improving awareness [9]; core developers can access in an efficient way all information that relates to a pull request and solicit opinions of the community ("crowd-source") about the merging decision.

A distributed development workflow is effective if pull requests are eventually accepted, and it is efficient if the time this takes is as short as possible. Advancing our insight in the effectiveness and efficiency of pull request handling is of direct interest to contributors and developers alike. The goal of this work is to obtain a deep understanding of pull request usage and to analyze the factors that affect the efficiency of the pull-based software development model. Specifically, the questions we are trying to answer are:

RQ1 How popular is the pull based development model?

RQ2 What are the lifecycle characteristics of pull requests?

RQ3 What factors affect the decision and the time required to merge a pull request?

RQ4 Why are some pull requests not merged?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568260

Our study is based on data from the Github collaborative development forge, as made available through our GHTorrent project [16]. Using it, we first explore the use of almost 2 million pull requests across all projects in Github. We then examine 291 carefully selected Ruby, Python, Java and Scala projects (in total, 166,884 pull requests), and identify, using qualitative and quantitative analysis, the factors that affect pull request lifetime, merging and rejection.

2. BACKGROUND

Since the appearance of the first open source implementations in 2001, distributed version control systems (DVCS), notably Git [8], have revolutionized the way distributed software development is carried out. Driven by pragmatic needs, most DVCSs were designed from scratch to work as advanced patch management systems, rather than versioned file systems, the then dominant version control paradigm. In most DVCSs, a file is an ordered set of changes, the serial application of which leads to the current state. Changes are stamped by globally unique identifiers, which can be used to track the commit's content across repositories. When integrating changes, the change sets can originate from a local filesystem or a remote host; tools facilitate the acquisition and application of change sets on a local mirror. The distributed nature of DVCSs enables a pull-based development model, where changes are offered to a project repository through a network of project forks; it is up to the repository owner to accept or reject the incoming pull requests.

The purpose of distributed development is to enable a potential *contributor* to submit a set of changes to a software project managed by a *core team*. The development models afforded by DVCSs are a superset of those in centralized version control environments [31, 6]. With respect to receiving and processing external contributions, the following strategies can be employed with DVCs:

Shared repository. The core team shares the project's repository, with read and write permissions, with the contributors. To work, contributors clone it locally, modify its contents, potentially introducing new branches, and push their changes back to the central one. To cope with multiple versions and multiple developers, larger projects usually adopt a *branching model*, i.e., an organized way to inspect and test contributions before those are merged to the main development branch [7].

Pull requests. 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 is then responsible to inspect the changes and pull them to the project's master branch. If changes are considered unsatisfactory, more changes may be requested; in that case, contributors need to update their local branches with new commits. Furthermore, as pull requests only specify branches from which certain commits can be pulled, there is nothing that forbids their use in the shared repository approach (*cross-branch pull requests*). An overview of the pull request process can be seen in Figure 1.

Pull Requests in Github. Github supports all types of distributed development outlined above; however, pull requests receive special treatment. The site is tuned to allow easy forking of projects by contributors, while facilitating the generation of pull requests through automatic comparison of project branches. Github's pull request model follows the generic pattern presented above; in addition it provides tools for contextual discussions and in-line code reviews. An example pull request on Github can be seen in Figure 2.

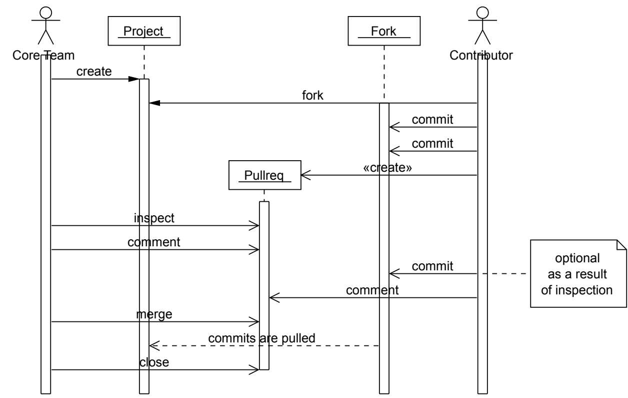

Figure 1: The pull request process.

A Github pull request contains a branch (local or in another repository) from which a core team member should pull commits. Github automatically discovers the commits to be merged and presents them in the pull request. By default, pull requests are submitted to the base ("upstream" in Git parlance) repository for inspection. The inspection is either a code review of the commits submitted with the pull request or a discussion about the features introduced by the pull request. Any Github user can participate to both types of inspection. As a result of the inspection, pull requests can be updated with new commits or be closed as redundant, uninteresting or duplicate. In case of an update, the contributor creates new commits in the forked repository, while Github automatically updates the displayed commits. The code inspection can then be repeated on the refreshed commits.

When the inspection process finishes and the pull requests are deemed satisfactory, the pull request 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 three ways, presented below sorted by the amount of preservation of the original source code properties:

1. Through Github facilities. Github can automatically verify whether a pull request can be merged without conflicts to the base repository. When a merge is requested, Github will automatically apply the commits in the pull request and record the merge event. All authorship and history information is maintained in the merged commits.

2. Using Git merge. When a pull request cannot be applied cleanly or when project-related policies do not permit automatic merging, a pull request can be merged using plain Git utilities, using the following techniques:

- *Branch merging:* The branch in the forked repository containing the pull request commits is merged into a branch in the base repository. Both history and authorship information are maintained, but Github cannot detect the merge in order to record a merge event [8, Chapter 3.2].
- *Cherry-picking:* Instead of merging all commits, the merger picks specific commits from the remote branch, which then applies to the upstream branch. The unique commit identifier changes, so exact history cannot be maintained, but authorship is preserved [8, Chapter 5.3].

A technique that complements both of the above is *commit squashing*: if the full history is not of interest to the project, several

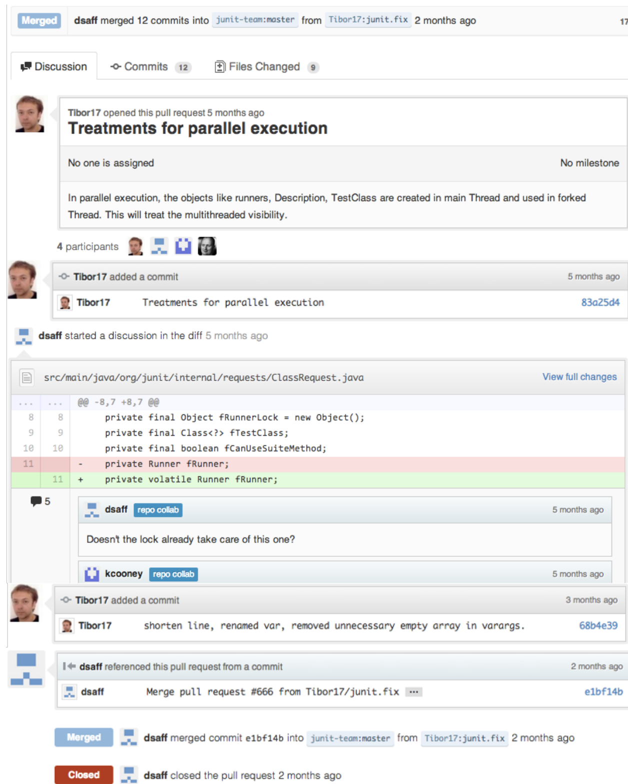

Figure 2: An example Github pull request (667 from junit-team/junit — edited for space). The participants first interact in a code review, the result of which is a new commit. The second reviewer then merges the pull request

consecutive commits are combined into a single one on the pull request branch, which can then be merged or cherry-picked to the upstream branch. In this case, the author of the commit is different from the person that applied the commit [8, Chapter 6.4]. Both cherry-picking and commit squashing are by-products of Git's support for re-ordering commits (*rebase*) [8, Chapter 3.6].

3. Committing the patch. The merger creates a textual difference between the upstream and the pull request branch, which she then applies to the upstream branch. Both history and authorship information are lost.

As the project branches are updated in a distributed manner, the changes in a pull request may interfere with new changes in the project's main branch. Merging such a pull request will result in *conflicts*. Github automatically detects conflicting pull requests and marks them as such. Conflicts can be resolved by either the contributor or a core team member; however, pull request etiquette dictates that the contributor takes care of bringing the pull request back into a state where there are no conflicts. The conflict resolution process involves pulling new commits from the project's main repository, making changes to eliminate the conflicts and extending the pull request with the resulting conflict eliminating commits.

Issues and pull requests are dual on Github; for each pull request, an issue is opened automatically. Commits can also be attached to issues to convert them to pull requests (albeit with external tools). This duality enables the core team to treat pull requests as work items, which can be managed using the same facilities used for issues. Moreover, issue discussions can include links to pull requests and vice versa, while specific commit message formats can be used to automatically close issues or pull requests when commits are merged to the project's main branch.

The open nature of Github's pull requests lends itself to a variety of usage patterns. Except from basic patch submission, pull requests can be used as a requirements and design discussion tool[1] or as a progress tracking tool towards the fulfillment of a project release.[2] In the first case, a pull request serves as a discussion board for soliciting the opinions of other developers while a new feature is being implemented. In the second case, pull requests are associated with milestones in Github's issue tracker.

3. RESEARCH DESIGN

The main focus of this study is to understand and explain how pull requests are used by projects to enable collaboration. To answer our research questions, we use a sequential mixed-methods approach, a procedure for collecting, analyzing, and integrating both quantitative and qualitative data at some stage of the research process within a single study for the purpose of gaining a better understanding of the problem [17]. For specific research questions, we first explore the domain quantitatively, and then highlight interesting cases by exploring cases qualitatively. Below, we present how we approached each research question.

RQ1 To assess the popularity of the pull-based development model, we provide and analyze descriptive statistics on the use of pull requests in Github. In particular, we investigate such questions as how many projects actually make use of pull requests, how many of the projects are original repositories (versus, e.g., forks), and how pull requests relate to Github's issue tracking facilities. The outcomes are presented in Section 5.

RQ2 and RQ3 Identifying the lifecycle characteristics of pull requests and determining the factors that affect them calls for a dedicated dataset of projects that have a sufficiently long history of using pull requests. This dataset is described in Section 4.2.

Given this dataset, we answer RQ2 and RQ3 by determining a set of suitable candidate features through consultation of related work in the fields of patch submission, bug triaging, code reviewing and distributed collaboration. Then, we clean it up through cross-correlation analysis to obtain a set of features with maximum predictive power. Using the data from the extracted features, we perform a detailed statistical analysis of pull request characteristics to answer RQ2 (Section 6).

Next, we use machine learning to retrieve the dominant features. Prior to running the classification algorithms, we automatically labeled each pull request with an outcome factor; in the case of the merge decision classification task, the label signifies whether the pull request has been merged. For the merge time task, we first filter out pull requests that have not been merged and then split the remaining data points into three classes (hour, day, more than a day) according to the time required to merge the pull request. The split points were chosen to reflect the results of RQ2, and split the available data points into roughly equally sized bins.

At a high level, the process to retrieve the dominant features for both classification tasks consists of two steps. First, we run each dataset through 6 classification algorithms, namely Random Forests (`randomforest`), variants of Logistic Regression (`logregr`) (binary for the merge decision task, multinomial for the merge time task) and Naïve Bayes (`naivebayes`), Support Vector Machines (`svm`), decision trees (`dtree`) and AdaBoost with decision trees (`adaboost`). We used those algorithms as they

[1] Github uses this internally: `https://github.com/blog/1124`

[2] `https://github.com/blog/831`

are known to perform well in large datasets [21] and have been used in previous work involving prediction models [13]. We do not perform any additional tuning to the classification algorithms. We only report results on the first three, as those performed best. Then, we select the best classifier and apply a classifier-specific process to rank features according to their importance in the classification process.

To evaluate the classification performance, we use the Accuracy (ACC) and Area Under the receiver operating characteristic Curve (AUC) metrics. To select the appropriate classification algorithm, we run a 10-fold random selection cross-validation and aggregate the mean values for each classification metric. At each iteration, the algorithm randomly samples half of the available data points, trains a classifier with 90% percent of the input and uses it to predict the remaining 10%. The 10-fold run results also allowed us to evaluate the metric stability across runs (Section 7).

RQ4 To examine why some pull requests are not merged, we qualitatively analyze a set of randomly chosen non-merged pull requests in depth. We use open coding (a grounded theory tool) to come up with an inclusive set of reasons of why pull requests are not merged as follows: the first author read the pull request discussion on Github for randomly selected pull requests and summarized the reasons for closing them into one sentence per sample; during a second pass, the descriptions were aggregated and codes were extracted. To validate the identified codes, all three authors applied them on a different set of pull requests, compared results, identified inconsistencies and retrofitted the initial selection of codes. The final set of codes was then applied on a third sample which we used to draw results from. The sampling process is described in Section 4.3.

4. DATA

4.1 Github Data

We used Github data as provided through our GHTorrent project [16], an off-line mirror of the data offered through the Github API. The Github API data come in two forms; a streaming data flow lists events, such as forking or creating pull requests, happening on repositories in real time, while a static view contains the current state of entities. To obtain references to the roots of the static view entities, the GHTorrent project follows the event stream. From there, it applies a recursive dependency-based parsing approach to yield all data offered through the API. The data is stored in unprocessed format, in a MongoDB database, while metadata is extracted and stored in a MySQL relational database. The GHTorrent dataset covers a broad range of development activities on Github, including pull requests and issues. The project has been collecting data since February 2012. Up to August 2013, 1.9 million pull requests from more than two hundred thousand projects have been collected.

4.2 Pull Request Project Sample

Project selection. To make the analysis practical, while avoiding to examine toy projects, we use a dataset consisting of all projects for which GHTorrent recorded more than 200 pull requests in the period between February 2012 and August 2013. The initial selection resulted in 374 projects. The following criteria were then applied to exclude projects from the initial selection:

- Projects should include tests. To measure the effect of testing on pull request acceptance, we could only use projects that include tests which we could measure reliably. For that, we exploited the convention-based project layout in the Ruby (Gem), Python, Java and Scala (both Maven) language ecosystems, so our project selection was limited to those languages.
- Projects should have at least one commit coming from a pull request, to ensure that the project is open to external contributions and that pull requests are not just used by developers inside the project.
- Projects should be developing software frameworks or applications, rather than documentation or programming languages. We excluded documentation projects, because we are interested in distributed software development. We excluded programming language implementation projects because we wanted to avoid cases where the developed programming language's core library was overshadowing the metrics of the actual implementation. This is especially true for the 5 Ruby implementations hosted on Github.

After selection, the full history (including pull requests, issues and commits) of the included projects was downloaded and features were extracted by querying the GHTorrent databases and analyzing each project's Git repository. Furthermore, for these selected projects we collected all merges and the values for all factors that we use in our machine learning experiment, as described below.

Merge detection. To identify merged pull requests that are merged outside Github, we resorted to the following heuristics, listed here in order of application:

1. At least one of the commits associated with the pull request appears in the target project's master branch.
2. A commit closes the pull request (using the `fixes:` convention advocated by Github) and that commit appears in the project's master branch. This means that the pull request commits were squashed onto one commit and this commit was merged.
3. One of the last 3 (in order of appearance) discussion comments contain a commit unique identifier, this commit appears in the project's master branch and the corresponding comment can be matched by the following regular expression:
   `(?:merg|appl|pull|push|integrat)(?:ing|i?ed)`
4. The latest comment prior to closing the pull request matches the regular expression above.

If none of the above heuristics identifies a merge, we mark the pull request as unmerged.

After creating the data files, we investigated projects where the pull request merge ratio was significantly less than the one we calculated across Github (73%), and in any case less than 40%, as this means that our heuristics are not good enough for this project. This way, we filtered out 2 projects, which we did not replace.

The final dataset consisted of 291 projects (99 Python, 91 Java, 87 Ruby, 14 Scala) and 166,884 pull requests (59,970; 55,468; 43,870 and 7,576 for Python, Ruby, Java and Scala projects respectively). Both distributions are representative of the contemporary popularity of each respective programming language on both Github and other sites.

Feature Extraction. The feature selection was based on prior work in the areas of patch submission and acceptance [24, 4, 32, 3], code reviewing [28], bug triaging [1, 14] and also on semi-structured interviews of Github developers [9, 26, 22]. The selected features are split into three categories:

Pull request characteristics. These features attempt to quantify the impact of the pull request on the affected code base. When examining external code contributions, the size of the patch is affecting both acceptance and acceptance time [32]. There are various metrics to determine the size of a patch that have been used by researchers: code churn [24, 27], changed files [24] and number of commits [11]. In the particular case of pull requests, developers reported that the presence of tests in a pull request increases their

confidence to merge it [26]. To investigate this, we split the churn feature into two features, namely `src_churn` and `test_churn`. The number of participants has been shown to influence the time to process of code reviewing [28]. Finally, through our own experience analyzing pull requests, we have found that in many cases conflicts are reported explicitly in pull request comments while in other cases pull requests include links to other related pull requests.

Project characteristics. These features quantify how receptive to pull requests the project is. If the project's process is open to external contributions, then we expect to see an increased ratio of external contributors over team members. The project's size may be a detrimental factor to the speed of processing a pull request, as its impact may be more difficult to assess. Also, incoming changes tend to cluster over time (the "yesterday's weather" change pattern [15]), so it is natural to assume that pull requests affecting a part of the system that is under active development will be more likely to merge. Testing plays a role in speed of processing; according to [26], projects struggling with a constant flux of contributors use testing, manual or preferably automated, as a safety net to handle contributions from unknown developers.

Developer. Developer-based features quantify the influence that the person who created the pull request has on the decision to merge it and the time to process it. In particular, the developer who created the patch has been shown to influence the patch acceptance decision [19]. To abstract the results across projects with different developers, we include features that quantify the developer's track record [9], namely the number of previous pull requests and their acceptance rate; the former has been identified as a strong indicator of pull request quality [26]. Bird et al. [5], presented evidence that social reputation has an impact on whether a patch will be merged; in our dataset, the number of followers on Github can be seen as a proxy for reputation.

All features are calculated at the time a pull request has been closed or merged, to evaluate the effect of intermediate updates to the pull request as a result of the ensuing discussion. Features that contain a temporal dimension in their calculation (e.g., `team_size` or `commits_on_files_touched`) are calculated over the three-month time period before the pull request was opened.

The initial selection contained 25 features. To check whether the selected features are sufficiently independent, we conducted a pair-wise correlation analysis using the Spearman rank correlation (ρ) metric across all features. We set a threshold of $\rho = \pm 0.7$, above which we eliminated features. Using this cutoff, we removed 2 features, `asserts_per_kloc` and `test_cases_per_kloc` as they where very strongly correlated ($\rho > 0.92$) with the included `test_lines_per_kloc` feature. We also removed features that could not be calculated reliably at the time a pull request was done (`followers` and `stars`). Finally, we merged similar features (i.e. `doc_files` and `src_files` were merged to `files_changed`).

The post processing phase left us with 15 features, which can be seen in Table 1. In general, very few features are correlated at a value $\rho > 0.2$, while only two, `src_churn` and `files_changed`, are strongly correlated at $\rho = 0.63$. While the correlation is strong, it is below our threshold and it is not definite; therefore we do not remove either feature from the dataset. All results are statistically significant ($n = 166,884, p < 0.001$).

4.3 Qualitative Data

To investigate how pull requests are used in practice and why some pull requests are not merged, we performed in-depth examination of random samples of pull requests followed by coding and thematic analysis. 100 pull requests were initially used by the first coder to identify discrete reasons for closing pull requests (bootstrapping sample), while a different set of 100 pull requests were used by all three coders to validate the identified categories (cross-validation sample). After cross validation, the two datasets were merged and a further 150 randomly selected pull requests were added to the bootstrapping sample to construct the finally analyzed dataset for a total of 350 pull requests.

5. POPULARITY OF PULL-BASED DEVELOPMENT

As of August 2013, Github reports more than 7 million repositories and 4 million users. However, not all those projects are active: in the period Feb 2012 — Aug 2013, the GHTorrent dataset captured events initiated by (approximately) 2,281,000 users affecting 4,887,500 repositories. The majority of registered repositories are forks of other repositories, special repositories hosting user web pages program configuration files and temporary repositories for evaluating Git. In the GHTorrent dataset, less than half (1,877,660 or 45%) of the active repositories are original repositories.

Pull requests are enabled by default on all repositories opened on Github; however, not all projects are using them to collaborate. In the period from February to August 2012, 315,522 original repositories received a single commit. From those, 53,866 (17%) received at least one pull request, while 54,205 (18%) used the shared repository approach, having received commits by more than one developers and no pull requests. The situation is similar during the same period in 2013; from the 1,157,625 repositories that received a single commit, 120,104 (10%) repositories received a pull request while 124,316 (11%) used the shared repository approach exclusively. In both cases, the remaining 65% and 79% are single developer projects. Across both years, 14% of the active repositories use pull requests. While pull request usage is increasing overall, partially reflecting Github's growth, the relative number of repositories using the pull request model has decreased slightly. An almost equal number of projects use pull requests and shared repositories for distributed collaboration.

For those projects that received pull requests in 2013, the mean number of pull requests per project is relatively low at 8.1 (median: 2, percentiles: 5%: 1, 95%: 21); however, the distribution of the number of pull requests in projects is highly skewed. Projects exists, such as Ruby on Rails and the Homebrew package manager, that have more that 5,000 pull requests. From the pull requests that have been opened in 2013, 73,07% have been merged using Github facilities, thereby indicating that pull requests in principle can work as a means for obtaining external contributions. Moreover, even though one might expect that it is the well known projects that receive most pull requests, this is only moderately supported by our data: the Spearman rank correlation between the number of stars of a project and the number of pull requests it has received is $\rho = 0.36$ ($p < 0.001, n = 239,131$).

Reviews on a pull request can either target the pull as whole or the individual commits, thereby resembling a code review. On average, each pull request receives 2,89 (quantiles: 5%: 0, 95%: 11, median: 1) discussion and code review comments. Even though any Github user can participate in the review process, usually it is the project community members that do so: only 0.011% of pull request comments come from users that have not committed to the project repository. Across projects that received pull requests in 2012, 35% also received a bug report (not pull-request based) on the Github issue tracker, indicating moderate use of Github's collaboration facilities by both the project and the project community.

Table 1: Selected features and descriptive statistics. Histograms are in log scale.

Feature	Description	5 %	mean	median	95 %	Histogram
Pull Request Characteristics						
`num_commits`	Number of commits in the pull request	1.00	4.47	1.00	12.00	
`src_churn`	Number of lines changed (added + deleted) by the pull request.	0.00	300.72	10.00	891.00	
`test_churn`	Number of test lines changed in the pull request.	0.00	88.88	0.00	282.00	
`files_changed`	Number of files touched by the pull request.	1.00	12.12	2.00	31.00	
`num_comments`	Discussion and code review comments.	0.00	2.77	1.00	12.00	
`num_participants`	Number of participants in the pull request discussion	0.00	1.33	1.00	4.00	
`conflict`	The word *conflict* appears in the pull request comments.	—	—	—	—	—
`forward_link`	Pull request includes links to other pull requests.	—	—	—	—	—
Project Characteristics						
`sloc`	Executable lines of code at pull request creation time.	1,390	6,0897	26,036	302,156	
`team_size`	Number of active core team members during the last 3 months prior to the pull request creation.	1.00	15.37	7.00	65.00	
`perc_ext_contribs`	The ratio of commits from external members over core team members in the last 3 months.	8.00	52.81	54.00	95.00	
`commits_files_touched`	Number of total commits on files touched by the pull request 3 months before the pull request creation time.	0.00	52.39	5.00	210.00	
`test_lines_per_kloc`	A proxy for the project's test coverage.	1.39	1,002.61	440.80	2,147.43	
Developer						
`prev_pullreqs`	Number of pull requests submitted by a specific developer, prior to the examined pull request.	0.00	45.11	14.00	195.00	
`requester_succ_rate`	% of the developer's pull requests that have been merged up to the creation of the examined pull request.	0.00	0.59	0.78	1.00	

RQ1: 14% of repositories are using pull requests on Github. Pull requests and shared repositories are equally used among projects. Pull request usage is increasing in absolute numbers, even though the proportion of repositories using pull requests has decreased slightly.

6. PULL REQUEST LIFECYCLE

Lifetime of pull requests. After being submitted, pull requests can be in two states: merged or closed (and therefore not-merged). In our dataset, most pull requests (84.73%) are eventually merged. This result is higher than the overall we calculated for Github; we attribute this to the fact that the dataset generation process employs heuristics to detect merges in addition to those happening with Github facilities.

For merged pull requests, an important property is the time required to process and merge them. The time to merge distribution is highly skewed, with the great majority of merges happening very fast. Measured in days, 95% of the pull requests are merged in 26, 90% in 10 and 80% in 3.7 days. 30% of pull requests are merged in under one hour; the majority of such pull requests (60%) come from the community, while their source code churn is significantly lower than that of the pull requests from the main team members (medians: 5 and 13 lines respectively). If we compare the time to process pull requests that are being merged against those that are not, we can see that pull requests that have been merged are closed much faster (median: 434 minutes) than unmerged (median: 2,250 minutes) pull requests. The results of an unpaired Mann-Whitney test ($p < 0.001$) showed that this difference is statistically significant, with a moderately significant effect size (Cliff's δ : 0.32). This means that pull requests are either processed fast or left lingering for long before they are closed.

Based on these observations, we check whether the pull requests originating from main team members are treated faster than those from external contributors. To answer it, we performed an unpaired Mann-Whitney test among the times to merge pull requests from each group. The result is that while the two groups differ in a statistically significant manner ($n_1 = 51,829, n_2 = 89,454, p < 0.001$), the apparent difference is negligible (Cliff's δ : -0.09). This means that merged pull requests received no special treatment, irrespective whether they came from core team members or from the community.

On a per project basis, if we calculate the median time to merge a pull request, we see that in the vast majority of projects (97%), the median time to merge a pull request is less than 7 days. The mean time to merge is not correlated with the project's size ($\rho = -0.05$), nor the project's test coverage ($\rho = 0.27$). It is however, strongly correlated ($\rho = -0.69$) with the contributor's track record: the more pull requests a developer has submitted to the same project, the lower the time to process each one of them. Moreover, projects are not getting faster at pull request processing by processing more pull requests; the correlation between the mean time to merge and the number of pull requests the project received is weak ($\rho = -0.23, n = 291, p < 0.01$).

Sizes of pull requests. A pull request bundles together a set of commits; the number of commits on a pull request is generally less than 10 (95% percentile: 12, 90% percentile: 6, 80% percentile: 3), with a median of 1. The number of files that are changed by a pull request is generally less than 20 (95% percentile: 36, 90% percentile: 17, 80% percentile: 7), with median number of 2. The number of total lines changed by pull requests is on average less than 500 (95% percentile: 1227, 90% percentile: 497, 80% percentile: 168) with a median number of 20.

Tests and pull requests. Except from the project's source code, pull requests also modify test code. In our sample, 33% of the pull requests included modifications in test code, while 4% modified test code exclusively. Of the pull requests that included modifications to test code, 83% were merged, which is similar to the aver-

age. This seems to go against the findings by Pham et al. [26], where interviewed developers identified the presence of tests in pull requests as a major factor for their acceptance. The presence of tests in a pull request does not seem to affect the merge time either: an unpaired Mann-Whitney test shows that while there is a statistically significant difference in the means of the pull request merge time between pull requests that include tests (median: 17 hours) and those that do not (median: 5 hours) ($pr_tests = 45,488, pr_no_tests = 95,980, p < 0.001$), the effect size is small ($\delta = 0.18$).

Discussion and code review. Once a pull request has been submitted, it is open for discussion until it is merged or closed. The discussion is usually brief: 95% of pull requests receive 12 comments or less (80% less than 4 comments). Similarly, the number of participants in the discussion is also low (95% of pull requests are discussed by less than 4 people). The number of comments in the discussion is moderately correlated with the time to merge a pull request ($\rho = 0.48, n = 141,468$) and the time to close a non-merged pull request ($\rho = 0.37, n = 25,416$).

Code reviews are integrated in the pull request process. While the pull request discussion can be considered an implicit form of code review, 12% of the pull requests in our sample have also been through explicit code reviewing, by having received comments on source code lines in the included commits. Code reviews do not seem to increase the probability of a pull request being merged (84% of reviewed pull requests are merged), but they do slow down the processing of a pull request: the unpaired Mann-Whitney test between the time required to merge reviewed (median: 2719 minutes) and non-reviewed (median: 295 minutes) pull requests gives statistically significant differences with a significant effect size ($\delta = 0.41$). Projects that employ code reviews feature larger code bases and bigger team sizes than those that do not.

Any Github user can participate in the discussion of any pull request. Usually, the discussion occurs between core team members trying to understand the changes introduced by the pull request and community members (often, the pull request creator) who explain it. In most projects, more than half of the participants are community members. This is not true however for the number of comments; in most projects the majority of the comments come from core team members. One might think that the bigger the percentage of external commenters on pull requests, the more open the project is and therefore the higher the percentage of external contributions; a Spearman test indicates that it is not true ($\rho = 0.22, n = 291, p < 0.05$).

RQ2: Most pull requests are less than 20 lines long and processed (merged or discarded) in less than 1 day. The discussion spans on average to 3 comments, while code reviews affect the time to merge a pull request. Inclusion of test code does not affect the time or the decision to merge a pull request. Pull requests receive no special treatment, irrespective whether they come from contributors or the core team.

7. MERGING AND MERGE TIME

To understand which factors affect the decision to merge and the time it takes to make this decision, we run the classification processes according to the method specified in Section 3. Each classifier attempts to predict the dependent variable (merge decision, merge time class) based on the features presented in Table 1. For the mergetime experiment, we excluded the `num_comments`, `num_commits` and `num_participants` features as they could not be measured at the pull request arrival time. Based on the results presented in Table 2, we selected the `randomforest` classification algorithm for both our experiments. For the mergetime experiment, `randomforest` achieved an AUC of 0.73, with a prior probability of 31%, 35% and 34% for each of the hour, day and more than a day classes respectively. For the mergedecision experiment, the prior probability for the dominant class was 84% which allowed the algorithm to achieve near perfect scores. In both cases, the stability of the AUC metric across folds was good (mergetime: $\sigma_{auc} = 0.019$, mergedecision: $\sigma_{auc} = 0.008$).

Table 2: Classifier performance for the merge decision and merge time classification tasks.

classifier	AUC	ACC	PREC	REC
mergedecision task($n = 166,884$)				
binlogregr	0.75	0.61	0.95	0.55
naivebayes	0.71	0.59	0.94	0.55
randomforest	0.94	0.86	0.93	0.94
merge time task($n = 141,468$)				
multinomregr	0.61	0.44	—	—
naivebayes	0.63	0.38	—	—
randomforest	0.73	0.59	—	—

To extract the features that are important for each classification task, we used the process suggested by Genuer et al. [12]. Specifically, we run the algorithm 50 times on a randomly selected sample of $n = 83,442$ items, using a large number of generated trees (2000) and trying 5 random variables per split. We then used the mean across 50 runs of the Mean Decrease in Accuracy metric, as reported by the R implementation of the random forest algorithm, to evaluate the importance of each feature. The results can be seen in Figure 3.

Finally, to validate our feature selection, we rerun the 10-fold cross-validation process with increasing number of predictor features starting from the most important one per case. In each iteration step, we add to the model the next most important feature. We stop the process when the mean AUC metric is within 2% from the value in Table 2 for each task. The selected set of features should then be enough to predict the classification outcome with reasonable accuracy, and therefore can be described as important [12].

For the mergedecision task, the feature importance result is dominated by the `commits_on_files_touched` feature. By re-running the cross validation process, we conclude that it suffices to use the features `commits_on_files_touched`, `sloc` and `files_changed` to predict whether a pull request will be merged (AUC: 0.94, ACC: 0.86). Therefore, we can conclude that the decision to merge a pull request is affected by whether it touches an actively developed part of the system (a variation of the "yesterday's weather" hypothesis), how large the project's source code base is and how many files the pull request changes.

For the mergetime task, there is no dominant feature; the classification model re-run revealed that at least 6 features are required to predict how fast a pull request will be merged. The classification accuracy was moderate (AUC: 0.74, ACC: 0.59), but still improved over random selection. The results provide evidence that the developer's previous track record, the size of the project and its test coverage and the project's openness to external contributions seem to play a significant role on how fast a pull request will be accepted.

RQ3: The decision to merge a pull request is mainly influenced by whether the pull request modifies recently modified code. The time to merge is influenced by the developer's previous track record, the size of the project and its test coverage and the project's openness to external contributions.

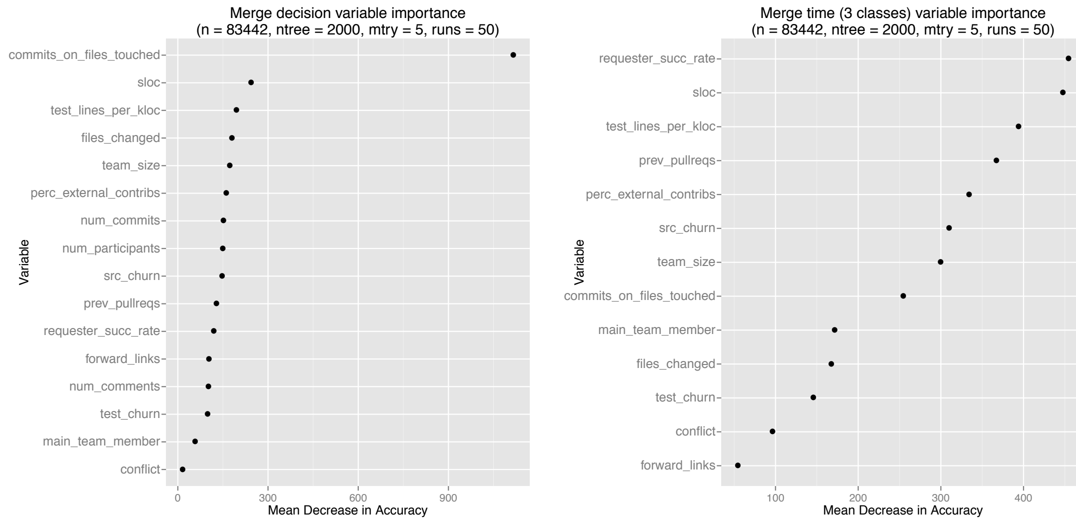

Figure 3: Random forest feature importance for predicting merge decision (a) and merge time (b)

Table 3: Reasons for closing pull requests without merging.

Reason	Description	%
obsolete	The PR is no longer relevant, as the project has progressed.	4
conflict	There feature is currently being implemented by other PR or in another branch.	5
superseded	A new PR solves the problem better.	18
duplicate	The functionality had been in the project prior to the submission of the PR	2
superfluous	PR doesn't solve an existing problem or add a feature needed by the project.	6
deferred	Proposed change delayed for further investigation in the future.	8
process	The PR does not follow the correct project conventions for sending and handling pull requests.	9
tests	Tests failed to run.	1
incorrect implementation	The implementation of the feature is incorrect, missing or not following project standards.	13
merged	The PR was identified as merged by the human examiner	19
unknown	The PR could not be classified due to lacking information	15

8. UNMERGED PULL REQUESTS

As most pull requests are indeed merged, it is interesting to explore why some pull requests are *not* merged. For that reason, we manually looked into 350 pull requests and classified the reasons in categories as described in Section 4.3. The cross-validation of the categories on a different set of pull requests revealed that the identified categories are enough to classify all reasons for closing a pull request, even though differences existed among the coders. The results are presented in Table 8.

The results show that there is no clearly outstanding reason for closing pull requests. However, if we group together close reasons that have a timing dimension (obsolete, conflict, superseded), we see that 27% of unmerged pull requests are closed due to concurrent modifications of the code in project branches. Another 16% (superfluous, duplicate, deferred) is closed as a result of the contributor not having identified the direction of the project correctly and is therefore submitting uninteresting changes. 10% of the contributions are rejected with reasons that have to do with project process and quality requirements (process, tests); this may be an indicator of processes not being communicated well enough or a rigorous code reviewing process. Finally, another 13% of the contributions are rejected because the code review revealed an error in the implementation.

Moreover, for 15% of the pull requests, the human examiners could not identify the cause of not merging them. This usually means that there was no discussion prior to closing the pull request or the pull request was automatically initiated and managed by external tools; incidentally, all but one project that had such pull requests in our random sample did not use Github's issue tracking facilities. Consequently, the use of pull requests by such projects may be superficial, only complementing parts of a process managed by other tools. Finally, the human examiner could identify 19% of the pull requests as merged even though the automated heuristics could not; this means that the proportion of merged pull requests reported in this study (84%) may be slightly underrated, due to non-inclusive heuristics. By extrapolation, the total number of merged pull requests could be as high as 90%.

It is interesting to note that only 13% of the contributions are rejected due to technical issues, which is the primary reason for code reviewing, while a total 53% are rejected for reasons having to do with the distributed nature of the pull request process (concurrent modifications) or the way projects handle communication of project goals and practices. This may mean that the pull-based model (or at least the way Github implements it) may be transparent for the project's core team [10] but not so much for potential contributors. The fact that human examiners could not understand why pull requests are rejected even after manually reviewing them supports this hypothesis further.

RQ4: 53% of pull requests are rejected for reasons having to do with the distributed nature of pull based development. Only 13% of the pull requests are rejected due to technical reasons.

9. DISCUSSION

9.1 The Pull-based Development Model

Development turnover. One of the promises of the pull request model is fast development turnover, i.e., the time between the submission of a pull request and its acceptance in the project's main repository. In various studies of the patch submission process in projects such as Apache and Mozilla, the researchers found that the time to commit 50% of the contributions to the main project repository ranges from a few hours [30] to less than 3 days [32, 3]. Our findings show that the majority (80%) of pull requests are merged within 4 days, 60% in less than a day, while 30% are merged within one hour (independent of project size). These numbers are indicating that pull-based development through pull requests may be more efficient than traditional email-based patches. Also, it is project-related factors that affect the turnover time, rather than characteristics of the pull request itself. This means that it is mostly up to the project to tune its processes (notably, testing coverage and process openess) for faster turnover.

Managing pull requests. The interviewees in Dabbish et al. [10] identify the management of pull requests as the most important project activity. Dabbish et al. mention that project managers "made inferences about the quality of a code contribution based on its style, efficiency, thoroughness (for example, was testing included?), and the submitter's track record". Some of the inspection points mentioned by project managers (testing code in pull requests, track record) are also included as features in our classification models, but they do not seem to affect the merge decision process as much. However, the developer track record is important for the speed of processing pull requests. Moreover, we found that from rejected pull requests, almost 53% are rejected due to the distributed nature of pull-based development. While the pull request process is transparent from the project manager's side (and praised for that by Dabbish et al.'s interviewees), our findings suggest it is less so from the potential contributor's point of view.

Attracting contributions. Pham et al. [26] mention that pull requests make casual contributions straightforward through a mechanism often referred to as "drive-by commits". As the relative cost to fork a repository is negligible on Github (54% of the repositories are forks), it is not uncommon for developers to fork other repositories to perform casual commits. Such commits might be identified as pull requests that contain a single commit from users that are not yet part of the project's community and comprise 7% of the total number of pull requests in 2012. Moreover, 3.5% of the forks were created for the sole purpose of creating a drive-by commit. More work needs to be done for the accurate definition and assessment of the implications of drive-by commits.

Crowd sourcing the code review. An important part of the contribution process to an open source project is the review of the provided code. Rigby and German [29], report that 80% of the core team members are also participating in the code reviews for patches, a number that is also in line with earlier findings by Mockus et al. [23]. In our dataset, we found that *all* core team members across all projects have participated in at least one discussion in a pull request. Moreover, we found that in *all* projects in our dataset, the community discussing pull requests is actually bigger than the core team members.

Democratizing development. One of the key findings of this work is that pull requests are not treated differently based on their origin; both core team members and external developers have equal chances to get their pull request accepted within the same time boundaries. Indeed, even the classification models we built assign to the corresponding feature low importance. In our opinion, this is a radical change in the way open source development is being carried out. Before pull requests, most projects employed membership promotion strategies [18] to promote interested third party developers to the core team. With pull requests, developers can contribute to any repository, without loss of authorship information. The chances that those contributions will get accepted are higher with pull requests; across Github, more than 70% of external contributions are merged (40% in other studies [29, 32]). Specialized sites such as Ohloh and CoderWall track developer activity and help developers advertise their expertise. We believe that the democratization of the development effort will lead to a substantially stronger commons ecosystem; this remains to be verified in further studies.

9.2 Implications

Contributors. Prospective project contributors want their contributions to be accepted. Our research shows that pull requests that affect parts of the project that have been changed often lately (are "hot") are very likely to get merged. Also 80% of the merged pull requests modify three or less files and include patches less than 100 lines long. Therefore, our advice to contributors seeking to add a particular feature or fix a bug is to "keep it short". If the contribution's purpose is to make the contributor known to the project community, it is also beneficial to affect a project area that is hot.

Core team. The primary job of the core team is to evaluate a list of pull requests and decide whether to apply them or not. To make sure pull requests are processed on time, the obvious strategy is to invest in a comprehensive test suite and ensure that the project processes are sufficiently open and transparent. To avoid development concurrency related issues, core team members could ask contributors to communicate their indented changes by opening an issue that is then augmented by code and converted to a pull request. The project should include a clear articulation of what is expected from a pull request, for example tests or localized changes, on a prominent location in the project's Github page.

A direct application of our results is the construction of *tools* to help the core team prioritize their work; since we can predict with very high accuracy whether a pull request will be merged or not, a potential tool might suggest which pull requests can be merged without further examination by the time they arrive. Other tools might examine the quality of the pull request at the contributor's site, and based on the project's profile, would provide automated suggestions for improvement (e.g., more tests, documentation).

9.3 Threats to Validity

Internal validity. Our statistical analysis uses random forests as a way to identify and rank cross-factor importance on two response variables. The classification scores in the mergetime case are not perfect, so feature ranking may not be exactly the same given a different dataset. Further work is needed on validating the models on data from different sources (e.g., Bitbucket) or projects in different languages.

To analyze the projects, we extracted data from i) the GHTorrent relational database ii) the GHTorrent raw database iii) each project's Git repository. Differences in the data abstraction afforded by each data source may lead to different results in the following cases: i) Number of commits in a pull request: During their lifecycle, pull requests may be updated with new commits. However, when developers use commit squashing, the number of commits is reduced to one. Therefore the number of commits metric used in our analysis is often an idealized version of the actual work that took place in the context of a pull request. ii) Number of files and commits on touched files: The commits reported in a pull request also con-

tain commits that merge branches, which the developer may have merged prior to performing his changes. These commits may contain several files not related to the pull request itself, which in turn affects our results. Therefore, we filtered out those commits, but this may not reflect the contents of certain pull requests.

External validity. In our study, we used merged data from several projects. The statistical analysis treated all projects as equal, even though differences do exist. For example, the larger project in our dataset, Ruby on Rails, has more than 7,000 pull requests while the smaller ones 200. While we believe that the uniform treatment of the samples led to more robust results in the classification experiment, variations in pull request handling among projects with smaller core teams may be ironed out. The fact that we performed random selection cross-validation (instead of the more common sliding window version) and obtained stable prediction results is, nevertheless, encouraging.

10. RELATED WORK

Arguably, the first study of DVCS systems as input for research was done by Bird et. al in [6]. One finding related to our work is that maintaining authorship information leads to better identification of the developer's contributions. Bird and Zimmermann [7] investigated the use of branches in DVCSs (in Section 2, we refer to this DVCS use as "shared repository") and found that excessive use of branching may have a measurable, but minimal, effect on the project's time planning. On the other hand, Barr et al. [2] find that branches offer developers increased isolation, even if they are working on inter-related tasks. Finally, Shihab et al. [31] investigate the effect of branching on software quality; they find that misalignment of branching structure and organizational structure is associated with higher post-release failure rates.

This work builds upon a long line of work on patch submission and acceptance. In reference [23], Mockus et al. presented one of the first studies of how developers interact on large open source projects in terms of bug reporting. Bird et al. [4] introduced tools to detect patch submission and acceptance in open source projects. Weißgerber et al. presented an analysis of patch submission, where they find that small patches are processed faster and have higher change to be accepted into the repository. Baysal et al. [3] find that 47% of the total patches make it into the source code repository, a number much lower than our finding for pull requests (84%). Jiang et al. [20] analyzed patch submission and acceptance on the Linux kernel, which follows the pull-based development model in a more decentralized and hierarchical manner. They find that the reviewing time is becoming shorter through time while contributors can reduce it by controlling, among others, the number of affected subsystems and by being more active in their community. Those findings are similar to ours, where we find that the contributor's previous track record and number of lines in the pull request affect the time to merge it.

An inherent part of pull-based development is peer-reviewing the proposed changes. In that sense, our work complements the work by Rigby and Bird [28] and supports many of their findings. Both works find that the peer review discussion is usually short, that peer-reviewed changes are small and that reviews happen before the changes are committed and are very frequent. Rigby and Bird's work examines industry-led projects and different code reviewing processes than ours. The fact that many aspects of the results are indeed similar leads us to hypothesize that it is the underlying process (pull-based development) that govern the reviewing process.

Recently, Github has been the target of numerous publications. Dabbish et.al [10] found that Github's transparency helps developers manage their projects, handle dependencies more effectively, reduce communication needs, and decide what requires their attention. Peterson [25] finds that open source software (OSS) development on Github works mostly similarly to traditional OSS development, with the exception of faster turnaround times. Pham et al. [26] examined the testing practices of projects on Github and found that the lower barriers to submission hinders high-quality testing as the work load on project member increases. Finally, McDonald and Goggins [22] find that increased transparency in pull requests allows allows teams to become more democratic in their decisions.

11. CONCLUSION

The goal of this work is to obtain a deep understanding of the pull-based software development model, as used for many important open source projects hosted on Github. To that end, we have conducted a statistical analysis of millions of pull requests, as well of a carefully composed set of hunders of thousands of pull requests from projects actively using the pull-based model.

Our main findings are as follows:

1. The pull-based model is not as popular as we had anticipated: Only 14% of the active projects use pull requests, but this number is equal to the number of projects using the shared repository approach (Section 5).
2. Most pull requests affect just a few dozen lines of code, and 60% are processed (merged or discarded) in less than a day. The merge decision is *not* affected by the presence of test code. Core members and external developers have equal chances to get their pull request accepted (Section 6).
3. The decision to merge is mainly affected by whether the pull request modifies recently modified code. The time to merge is influenced by various factors, including the developer's track record, and the project's test coverage.
4. 53% of non-merged pull requests are rejected for reasons related to the distributed nature of pull-based development. Only 13% of the pull requests are rejected due to technical reasons.

Our findings have the following implications:

1. The pull-based model calls for a revision of some of our current understanding of open source software development. Interesting research directions might include the formation of teams and management hierarchies, novel code reviewing practices and the motives of developers to work in a highly transparent workspace.
2. Teams seeking to attract external contributors and to speed up merging of contributions can do so not just by providing clear pull request processing guidelines, but also by incorporating a high coverage test suite.
3. Insufficient task articulation seems the most important cause for wasted (non-merged) work: Devising new ways for integrating task coordination into the pull-based model is a promising area of further research.

Last but not least, our dataset provides a rich body of information on open source software development. The dataset as well as custom-built Ruby and R analysis tools are available on the Github repository gousiosg/pullreqs, along with instructions on how to use them.

12. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments. This work is partially supported by the Marie Curie IEF 298930 — SEFUNC and the NWO 639.022.314 — TestRoots projects.

13. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of ICSE '06*, pages 361–370. ACM, 2006.

[2] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *Proceedings of FASE '12*. Springer, 2012.

[3] O. Baysal, R. Holmes, and M. W. Godfrey. Mining usage data and development artifacts. In *Proceedings of MSR '09*, pages 98–107. IEEE, 2012.

[4] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *Proceedings of MSR '07*, page 26. IEEE Computer Society, 2007.

[5] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? Immigration in open source projects. In *Proceedings of MSR '07*, page 6. IEEE Computer Society, 2007.

[6] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining Git. In *Proceedings of MSR '09*, pages 1–10, 2009.

[7] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of FSE '12*, pages 45:1–45:11. ACM, 2012.

[8] S. Chacon. *Pro Git*. Expert's Voice in Software Development. Apress, 1rst edition, Aug 2009.

[9] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in Github: transparency and collaboration in an open software repository. In *Proceedings of CSCW '12*, pages 1277–1286. ACM, 2012.

[10] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Leveraging transparency. *IEEE Software*, 30(1):37–43, 2013.

[11] B. Fluri, M. Wursch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Soft. Eng.*, 33(11):725–743, 2007.

[12] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot. Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225 – 2236, 2010.

[13] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *In Proceedings of ESEM '12*, pages 171–180. ACM, 2012.

[14] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *In Proceedings of RSSE '10*, pages 52–56. ACM, 2010.

[15] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM '04*, pages 40 – 49, sept. 2004.

[16] G. Gousios. The GHTorrent dataset and tool suite. In *Proceedings of MSR '13*, May 2013.

[17] N. V. Ivankova, J. W. Creswell, and S. L. Stick. Using mixed-methods sequential explanatory design: From theory to practice. *Field Methods*, 18(1):3–20, 2006.

[18] C. Jensen and W. Scacchi. Role migration and advancement processes in OSSD projects: A comparative case study. In *Proceedings of ICSE '07*, pages 364–374. IEEE Computer Society, 2007.

[19] G. Jeong, S. Kim, T. Zimmermann, and K. Yi. Improving code review by predicting reviewers and acceptance of patches. *Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO)*, 2009.

[20] 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 MSR '13*, pages 101–110. IEEE Press, 2013.

[21] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008.

[22] N. McDonald and S. Goggins. Performance and participation in open source software on github. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 139–144. ACM, 2013.

[23] 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.*, 11(3):309–346, 2002.

[24] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of ICSE '05*, pages 284–292. ACM, 2005.

[25] K. Peterson. The github open source development process. Technical report, Mayo Clinic, May 2013.

[26] 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 ICSE '13*, pages 112–121. IEEE Press, 2013.

[27] J. Ratzinger, M. Pinzger, and H. Gall. EQ-mine: predicting short-term defects for software evolution. In *Proceedings of FASE '07*, pages 12–26. Springer-Verlag, 2007.

[28] P. C. Rigby and C. Bird. Convergent software peer review practices. In *Proceedings of FSE '13*, 2013.

[29] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. *University of Victoria, Canada, Tech. Rep. DCS-305-IR*, 2006.

[30] 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 ICSE '08*, pages 541–550. ACM, 2008.

[31] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *In Proceedings of ESEM '12*, pages 301–310. ACM, 2012.

[32] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *Proceedings of MSR '08*, pages 67–76. ACM, 2008.