SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Special Focus on Software Engineering for Internet Computing

August 2016, Vol. 59 080104:1–080104:14
doi: 10.1007/s11432-016-5595-8

Determinants of pull-based development in the context of continuous integration

Yue YU[1,2]*, Gang YIN[1,2], Tao WANG[1,2], Cheng YANG[1,2] & Huaimin WANG[1,2]

[1]*College of Computer, National University of Defense Technology, Changsha* 410073*, China;*
[2]*National Laboratory for Parallel and Distributed Processing, Changsha* 410073*, China*

Received April 25, 2016; accepted May 30, 2016; published online July 18, 2016

Abstract The pull-based development model, widely used in distributed software teams on open source communities, can efficiently gather the wisdom from crowds. Instead of sharing access to a central repository, contributors create a fork, update it locally, and request to have their changes merged back, i.e., submit a *pull-request.* On the one hand, this model lowers the barrier to entry for potential contributors since anyone can submit *pull-requests* to any repository, but on the other hand it also increases the burden on integrators, who are responsible for assessing the proposed patches and integrating the suitable changes into the central repository. The role of integrators in pull-based development is crucial. They must not only ensure that pull-requests should meet the project's quality standards before being accepted, but also finish the evaluations in a timely manner. To keep up with the volume of incoming pull-requests, continuous integration ($\mathcal{CI}$) is widely adopted to automatically build and test every pull-request at the time of submission. $\mathcal{CI}$ provides extra evidences relating to the quality of pull-requests, which would help integrators to make final decision (i.e., accept or reject). In this paper, we present a quantitative study that tries to discover which factors affect the process of pull-based development model, including acceptance and latency in the context of $\mathcal{CI}$. Using regression modeling on data extracted from a sample of GitHub projects deploying the Travis-CI service, we find that the evaluation process is a complex issue, requiring many independent variables to explain adequately. In particular, $\mathcal{CI}$ is a dominant factor for the process, which not only has a great influence on the evaluation process *per se*, but also changes the effects of some traditional predictors.

Keywords pull-request, continuous integration, GitHub, distributed software development, empirical analysis

Citation Yu Y, Yin G, Wang T, et al. Determinants of pull-based development in the context of continuous integration. Sci China Inf Sci, 2016, 59(8): 080104, doi: 10.1007/s11432-016-5595-8

1 Introduction

Software development process plays a key role in software engineering [1]. There are many important benefits to be gained from building up a mature software process [1, 2], especially for reducing costs and improving software quality. Open source software development exploits the distributed intelligence of participants in internet communities [3]. Currently, the pull-based development model [4] is widely used in distributed software teams to integrate incoming changes into a project's codebase [5, 6]. It is an efficient model for software projects to gather the wisdom from crowds (i.e., the large group of

* Corresponding author (email: yuyue@nudt.edu.cn)

external contributors compared to the core team). Enabled by `git`, the distributed version control system, pull-based development provides a chance that contributors to a open source project need not share access to a central repository anymore. Instead, anyone can create *forks* (i.e., local clones of the central repository), update them locally and, whenever ready, request to have their changes merged back into the main branch by submitting a *pull-request*. Compared to traditional collaboration development models in open source [7,8] (e.g., patch submission and acceptance via mailing lists or bugzilla), the pull-based model offers great advantages in terms of process automation. Nowadays, the modern collaborative coding platforms, e.g., GitHub, BitBucket, Gitorius, provide integrated functionality for pull-request generation, notification, in-line code review, contextual discussion, automatic testing, and merger. The popularity of pull-based development model is constantly increasing [5,9]. On GitHub alone, almost half of the collaborative projects use the pull-request mechanism and received more than 440000 pull-requests per month in 2015 on average. Moreover, more and more contributions, irrespective of whether they come from core team with write access to the repository or from outside contributors, are submitted as pull-requests, ensuring that only reviewed code gets merged [5, 6, 9]. Additionally, the pull-request mechanism can be integrated with multiple social media features [10–14]. GitHub users can freely *watch* the development activities of popular repositories, *follow* distinguished developers and *comment* others' contributions. Thus, pull-requests are extended to many scenarios beyond basic patch submission, e.g., conducting code reviews and discussing new features [6].

While the pull-based model offers a much lower barrier to entry for potential contributors, it also increases the burden on core team members who decide whether to integrate them into the main development branch [6, 15]. The role of integrators in pull-based development is crucial [6, 11]. They must ensure not only that pull-requests are evaluated in a timely matter and eventually accepted, to secure the project's growth, but also that all contributions meet the project's quality standards. In large projects, the volume of incoming pull-requests is quite a challenge [6, 16]. On the one hand, prioritizing pull-requests is one of the main concerns of integrators in their daily work [6, 11]. On the other hand, many contributors are suffering from the delayed feedback of their pull-requests, as reported in the survey [9].

To reduce human workload, GitHub synthesizes the *Continuous Integration* ($\mathcal{CI}$) to support the process automation of pull-request [15, 17]. Whenever a new pull-request is received by a project using $\mathcal{CI}$, it is triggers a $\mathcal{CI}$ step that all the new changes will be merged with the latest code of main branch, then automatically build and run existing test suites to detect the potential bugs. Based on the $\mathcal{CI}$ outcomes (e.g., pass or failure), the integrators present suggestions to the submitter (e.g., how to fix failures) or make a decision (i.e., accept or reject the pull-request). Theoretically, $\mathcal{CI}$ acts as a gatekeeper assisting integrators to gain productivity and guard the code quality.

In this paper, we report on a quantitative study that tries to resolve which factors affect the evaluating process of $\mathcal{CI}$-embedded pull-request mechanism in GitHub. Starting from our previous work [18], we further analyze pull-request acceptance and latency associated with $\mathcal{CI}$-related metrics by controlling for other confounds. In summary, the contributions of this paper are:

- We claim that the evaluation of pull-request is a complex issue, requiring many independent variables to explain adequately. We construct a comprehensive study containing the measures in project-level, pull-request level, submitter-level, workflow and $\mathcal{CI}$-related level.
- We find that the presence of $\mathcal{CI}$ is a dominant factor for both pull-request acceptance and latency. In particular, the pull-requests with $\mathcal{CI}$ failures have about 89.6% greater odds of being rejected eventually; for pull-request latency, the $\mathcal{CI}$ running time is highly significant and covers more than 21% of the variance explained.
- We find that the the size of a pull-request, the number of comments and the submitter's social properties are strong factors influencing the pull-request process in expected ways. Open source projects prefer (i.e., accept quickly) a good pull-request with small size, less controversy and originated from an acquainted and trusted contributor.
- Interestingly, we find that technical factors (e.g., test inclusion and hotness) exert different influences on pull-request acceptance versus latency. For example, the odds of acceptance increases by 101.4%, when the pull-requests associated with "hot" files in a given project (i.e., very strong and significant positive

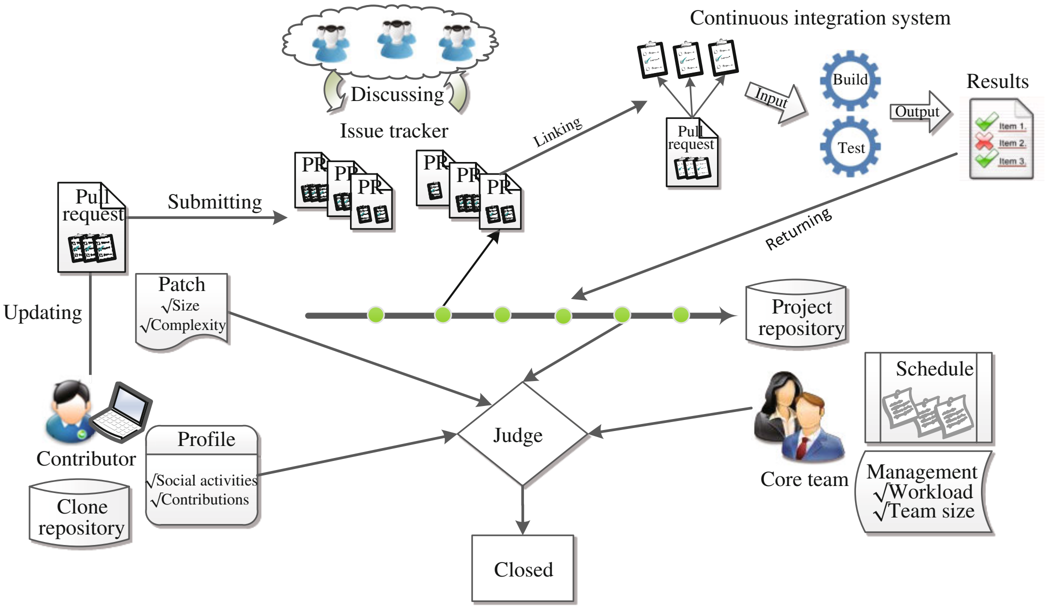

Figure 1 (Color online) Overview of the process of $\mathcal{CI}$-embedded pull-request

effect on pull-request acceptance). However, those pull-requests may need a longer evaluation with a very small but significant effect size.

The remainder of this paper is structured as follows: Section 2 presents more background, related work and our research questions. Section 3 illustrates the research methods. Our results and discussion can be found in Section 4. Finally, we draw the conclusions in Section 5.

2 Background

2.1 Continuous integration and pull-request

Arguably, *Continuous Integration* ($\mathcal{CI}$) originated from the imperatives of agility [19], aiming to responding quickly to customer requirements without compromising software quality. The basic notion [20, 21] of $\mathcal{CI}$ is that all developers' work within a team is continually compiled, built, and tested. This process is a perpetual check on the quality of contributed code, and mitigates the risk of "breaking the build" at the last moment, because of major, incompatible changes by different people or different sub-teams. Without $\mathcal{CI}$ process, software is considered broken until proven to work, typically during a testing or integration stage. With $\mathcal{CI}$ process, assuming a comprehensive automated test suite, software is proven to work with every new change, and serious regressions can be detected and fixed immediately. Also, it can be viewed as a type of process automation [17]. Supporting by $\mathcal{CI}$, automated services are embedded into the development environment to carry out the quality inspection continually and automatically, rather than wait for some sort of human-controlled gate-keeping of code. In the context of distributed, globalized development, cultural, geographical and time differences raise the specter of process variation and non-repeatability, and thus amplify the imperatives to adopt process automation. This applies even more strongly to open source software projects where, in addition to the above issues, volunteers are involved, and there is also a lack of centralized control [22–24]. Especially, the pull-based development model further to democratize contributing to open source projects, because anyone can contribute to any repository via submitting pull-requests, even if they are not part of the development team.

Integrating with $\mathcal{CI}$, the process of pull-request is depicted in Figure 1. First of all, a potential contributor forks (i.e., clone) an interesting project repository, and contribute code locally either by creating a new feature or by fixing some bugs. When the work is done, the contributor sends the changes from the clone repository to original repository packaged as a pull-request. At this point, the $\mathcal{CI}$ service kicks in. The pull-request is automatically merged with the latest code of the main branch into a new testing branch. Then, $\mathcal{CI}$ builds the whole project, and runs the existing test suites to detect the potential bugs. After automatic testing, the integrators can start a targeted review process by focusing on the critical parts associated with the outcomes of $\mathcal{CI}$ (e.g., the failed test cases). Typically, if tests fail, the pull-request would be rejected by one of the integrators. Meanwhile, the reviewers may also comment on why the contribution is inappropriate and how it can be improved. If tests pass, core team members proceed to do a team-wide code review by commenting inline on (parts of) the code. Resorting to the open-ended issue tracker, all users in GitHub, during above stage, have the chance to review that pull-request. If necessary, the submitter would be requested for modifications by updating the pull-request with some more new code. Whenever the new changes are submitted, $\mathcal{CI}$ will build and test the whole project again. After a cycle of comments, revisions and automatic building and testing, if everyone is satisfied, the pull-request is closed and merged.

Nowadays, $\mathcal{CI}$ is widely used on GitHub, and numerous tools that support $\mathcal{CI}$ exist [25]. There are 75% of projects that use pull-requests heavily also use $\mathcal{CI}$, either in hosted services (e.g., Travis-CI [26]) or in standalone setups [6].

2.2 Pull-request evaluation

As shown in Figure 1, evaluating $\mathcal{CI}$-embedded pull-requests is a complex iterative process affected by many factors (e.g., patch size, social connection and $\mathcal{CI}$ outcomes) and involving multiple stakeholders (e.g., submitter, core team and $\mathcal{CI}$ service).

Recently, researchers have started investigating the pull-request workflow in GitHub [5, 6, 12, 13, 16, 27, 28], trying to understand which factors influence the evaluation process of pull-request (e.g., decision to merge). However, (i) these effects are not yet well understood (e.g., results from different studies often diverge, as discussed below); and (ii) pull-request evaluation has not been studied in the context of $\mathcal{CI}$.

Using machine learning techniques, Gousios et al. [5] report that pull-requests touching actively developed (hot) parts of the system are preferentially accepted. The submitter's track record, the project size and its test coverage are the most prominent predictors for pull-requests' merge time (i.e., latency). In a follow-up qualitative study, Gousios et al. [6] find that target area hotness, existence of tests, and overall code quality are recognized as important determinants of pull-request acceptance by GitHub integrators taking part in their survey. In the same survey [6], the authors also asked survey respondents to: (i) rate several predefined factors by perceived importance to pull-request latency; and (ii) list additional determinants of latency, if any. The following ranking of predefined factors emerged (in decreasing order): project area hotness, existence of tests in the project, presence of tests in the pull-request, pull-request churn (i.e., number of changed lines), the submitter's track record, number of discussion comments, and number of commits. Among the many other self-identified factors, only reviewer availability and pull-request complexity are recognized by at least 5% of their respondents. These factors, perceived to be important, differ significantly from those data mined in the prior machine learning experiment (three classes: merged within one hour; one day; the rest) by the same authors [5]. Extending above insights, Hellendoorn et al. [28] confirm that the stylistic properties (relating to "naturalness" [29]) of submitted code would affect the review process of pull-request by a quantitatively study. They find that: (i) the merged pull-requests on GitHub are significantly conformance to project codebase from a language model perspective compared to those rejected; and (ii) less similar pull-requests are more likely to be subject thorough human review.

In contrast, GitHub provide transparent work environments [10, 11] for developers, which is benefit for innovation, knowledge sharing and community building. The transparency allows project managers to evaluate pull-requests utilizing a wealth of social information in submitters' profile pages. Based on results from a regression modeling study, Tsay et al. [16] argue that the strength of the social connection between the submitter and integrators is the determining factor of pull-request acceptance, instead of technical factors. To analyze the social connection among developers in GitHub, Yu et al. [12–14] present some typical social structures and behavior patterns by mining *follow-networks* [14] and *comment-*

networks [12]. They also design a mixed approach [13] that interpolating social relations with technical expertise to recommend reviewers to pull-requests.

2.3 Research questions

In this paper, we focus on both pull-request acceptance and latency, which are the understudied and arguably most important problems of pull-request process. Even though prior work [5,6,16,27] has identified several social and technical factors that influence pull-request evaluation, as discussed in Subsection 2.2, we believe the intricacies of this process are not yet accurately captured. Furthermore, there is recurring support for the importance of process factors to team outcomes in the empirical software engineering literature, e.g., [30, 31]. Focusing on the integrator workflow, we expect a multitude of process-related factors to influence pull-request process, e.g., current workload and availability of reviewers, pull-request description quality (akin to the quality of issue reports said to impact the issues' resolution time [32,33]), and the integrators' responsiveness (in recent work [34] we found that pull-request submitters may become discouraged by unresponsive integrators, leading to communication breakdown and conflict). Still, to replicate previous results and establish a baseline compared with $\mathcal{CI}$ inclusion, we ask:

RQ1. How do social, technical and process-related factors influence pull-request acceptance and latency?

Furthermore, $\mathcal{CI}$ has already played an essential role on pull-based development. The general literature on $\mathcal{CI}$ suggests that the continuous application of quality control checks speeds up overall development, and ultimately improves software quality [19]. On GitHub, the integration of $\mathcal{CI}$ in the pull-request workflow should enable project members to find integration errors more quickly, tightening the feedback loop. Based on reported widespread use of $\mathcal{CI}$ and the importance of testing related factors [6,15,16], we expect that $\mathcal{CI}$ will play a prominent role in the pull-request evaluation process, enabling integrators to better deal with general issues of scale. We ask:

RQ2. What is the effect of $\mathcal{CI}$ on pull-request acceptance and latency?

Lastly, the evaluation process of integrators is improved a lot after $\mathcal{CI}$ adoption, e.g., integrators only need focusing on code style and leave the quality (e.g., buggy code) inspection to $\mathcal{CI}$ automatic tools, if the project has sufficient test suites. Thus, the influence of traditional factors on pull-request process would be changed, after adding $\mathcal{CI}$-related predictors. We ask:

RQ3. What is the difference of traditional predictors affected by $\mathcal{CI}$-related variables involving?

3 Methods

3.1 Dataset

In this paper, we composed a dataset of representative open source projects that made heavy use of $\mathcal{CI}$-embedded pull-requests mechanism in GitHub. Our data collection process started form the *GHTorrent* [35] dump dated 2014-11-10. There were two stages for preparing our dataset, i.e., projects selection and pull-request completion.

First of all, we identified candidate original (i.e., non-forked) projects which contained at least 1000 pull-requests in total. We focus on the 5 most popular languages on GitHub[1], i.e., JavaScript, Python, Ruby, Java and C++. Then, we checked every project whether it used $\mathcal{CI}$ service or not. Here, we focused on two popular $\mathcal{CI}$ services, i.e., *Travis-CI* and *Jenkins-CI*, which are wildly adopted by projects in GitHub [17,25]. Using *Travis-CI*, all the build&testing processes of projects are hosted on a remote build server. After that, the GitHub pull-request UI is updated automatically with CI results (i.e., failure, error or pass in the build&testing process). Therefore, all the projects using it operate in the same context. The entire build history of projects using *Travis-CI* is available through its API[2]. In contrast, *Jenkins-CI* is a self-hosted system, so that projects set up their $\mathcal{CI}$ service locally. Thus, the $\mathcal{CI}$ virtual

[1] http://githut.info/

[2] http://docs.travis-ci.com/api/

Table 1 Basic statistics for our dataset

Language	#PRs received	#PRs merged	#PRs CI_tested
Ruby	28409	20755	19790
Python	28903	24039	14787
JavaScript	26983	17920	18816
Java/C++	18989	13456	7647
Total	103284	76170	61040

environment for each project is from individual to individual. In addition, only data on recent builds is stored for public accessing. Due to the limitations of *Jenkins-CI*, we restricted our attention in this study to the projects using *Travis-CI*. Finally, we combine Java with C++ as traditional language compared to other scripting languages, then select top 10 projects according to the number of received pull-requests under each programming language to generate a relatively balanced dataset.

In the second stage, we further extracted meta-data (e.g., number of comments) about all pull-requests. We ignored pull-requests that are still open and gathered the major part of meta-data from *GHTorrent* received after January 1, 2010. The missing data of *GHTorrent*, e.g., description and actual contents (diffs), was collected via GitHub API[3]. Next, we combined each pull-request with the corresponding *Travis-CI* logs, including the $\mathcal{CI}$ timestamps and build&testing results of all commits.

After above preprocessing, our dataset contains 103284 pull-requests collected from 40 different projects in GitHub, as summarized in Table 1. In our dataset, there are 73.7% of pull-requests have been merged by core teams, and 59.1% of pull-requests have been diagnosed by the cloud-based $\mathcal{CI}$ service (i.e., *Travis-CI*).

3.2 Measures

3.2.1 *Outcomes*

Our goal was to determine the factors that influenced the process of pull-based development model (i.e., pull-request acceptance and latency). The two outcome measures are:
PR_merged: binary, whether the PR has been merged into the project's code base.
PR_latency: the time in minutes interval between pull-request creation and closing date. In case of "reopened" pull-requests, we only considered the date when they were first closed.

3.2.2 *Predictors*

Our prior work [18] shows that the pull-request latency requires many independent variables to explain adequately. In this paper, we classified our metrics into project-level, pull-request level, submitter-level, workflow and $\mathcal{CI}$-related metrics, as discussed in Section 2.

Project-level Metrics:
project_age: the time from project creation on GitHub to the pull-request creation in months, which is used as a proxy for maturity. Mature projects are likely to have different contribution dynamics.
team_size: the number of active integrators, who decide whether to accept the new contributions, during the three months prior to pull-request creation. Larger teams may be better prepared to handle higher volumes of incoming pull-requests. We used a heuristic method to estimate the integrators for a given project. Since the authentication limitation[4] of GitHub (i.e., only users with access to the repository can retrieve the list of core developers), we identified the integrators as those core members that can close others' pull-requests or issues. Because an integrator would have several GitHub id, we utilized the emails, login names and user names to perform identity merging [36].
area_hotness: the median of the number of commits to files touched by the pull-request relative to all project commits during the last three months. It is interesting to know the contribution process of

[3] https://developer.github.com/v3/
[4] https://developer.github.com/v3/repos/collaborators/#list

the "hot" area in a project. For example, do integrators need more time to evaluate the pull-requests concerning to the "hot" area to avoid conflicts? Are those pull-requests more likely to be accepted or rejected?

Pull-request level Metrics:

churn: total number of lines added and deleted by the pull-request. Bigger code changes may be more complex, and require longer code evaluation.

n_commits: total number of commits in the pull-request, which is a signal of pull-request size. *Travis-CI* tests each commit separately.

description_complexity: total number of words in the pull-request title and description. Longer descriptions may indicate higher complexity (longer evaluation), or better documentation which can be facilitate evaluation process akin to issue reports [32].

test_inclusion: binary, measuring if the pull-request touched at least one test file. We identified test files using the conversions of folders organization and file names extension on GitHub [16, 37, 38]. For example, files are consider as test files if they were located at the "*/test/*" or "*/spec/*" folders, or their filenames contain the word "test" as prefix or suffix. Previous studies [5, 6, 16] have pointed out that integrators prefer pull-requests containing tests (i.e., the pull-requests coming with test are more likely to be accepted).

Submitter-level Metrics:

core_team: binary, whether the submitter is member of core development team. There is a recognized structure, i.e., "core" and "periphery", with open source software projects. Core developers who make contributions to their own project would be more likely to have their contributions accepted by fellow project integrators.

social_strength: the fraction of team members that interacted with the submitter in the last three months. Integrators may favor and trust contributors more strongly connected to them. We modified the our prior work of *comment network* [12, 13] to measure social relations among developers in a given project. If a contributor c has co-occurred with the project integrator m in at least one discussion of issue, commit or pull-request during last three months, we link c with m by the edge e_{cm}. In a given project, the strength of social connection between the contributor c and the core team can be calculated by (1).

$$social\_strength_c = \frac{\#linked\_integrators}{|core\_team|} \in [0, 1]. \tag{1}$$

n_followers: the number of GitHub users following the incoming pull-request's submitter at the time of creation, as a measure of social reputation. We expect that the pull-requests originated from contributors with high reputation are more likely to be accepted, and also fast.

Workflow Metrics:

n_comments: total number of overall and inline comments part of a pull-request discussion. Pull-requests with lots of comments tend to signal controversy [10].

first_human_response: the time interval in minutes from pull-request creation to the first response by reviewers, which can be indicated as a measure of the project team's responsiveness.

workload: total number of pull-requests still open in each project at current pull-request creation time. It is reasonable to assume that if the workloads of integrators are heavy, the new received pull-request would need more time to be processed.

integrator_availability: the minimum number of hours ($0 \ldots 23$) until either of the top two integrators (by number of pull-requests handled the last three months) are active, on average (based on activity in the last three months), during 24 h. Two reviewers find an optimal number of defects during code review [39, 40], hence our choice for top two.

#issue_tag and *@mention_tag*: binary variables to encode the presence of "#" tags (links to issues) and "@" tags (to notify integrators directly) in the pull-request title or description. If a pull-request contains those social tags, it would be more explicit for evaluation, i.e., who should review it (*@mention_tag*); which issue it solved (*#issue_tag*).

Table 2 Summary of our dataset after removing outliers

Statistic	Mean	St. Dev.	Min	Median	Max
PR_merged	0.77	0.42	0	1	1
PR_latency	12586.17	46762.55	0.03	942.52	1126255.00
proj_age	36.81	17.87	3	35	81
team_size	13.15	7.45	1	12	34
area_hotness	0.02	0.04	0.00	0.01	0.69
churn_additions	72.55	163.32	0	14	1463
churn_deletions	25.64	67.40	0	3	635
n_commits	3.02	7.01	1	1	290
description_complexity	45.272	71.343	2	23	990
test_inclusion	0.29	0.45	0	0	1
core_team	0.55	0.50	0	1	1
social_strength	0.40	0.30	0.00	0.40	1.00
n_followers	29.53	50.63	0	10	315
n_comments	4.97	9.00	0	2	223
first_human_response	934.44	2966.45	0.00	22.66	30245.07
workload	47.31	58.85	0	27	296
integrator_availability	2.80	3.88	0	1	22
@mention_tag	0.08	0.27	0	0	1
#issue_tag	0.24	0.43	0	0	1
Friday_effect	0.15	0.36	0	0	1
$\mathcal{CI}$_failure	0.22	0.42	0	0	1
$\mathcal{CI}$_latency	2630.50	8611.88	0.33	35.35	83437.13

Friday_effect: binary, true if the pull-request arrives on Friday. Prior research [41] shows that the likelihood is highest on Friday that the code patches would introduce bugs.

$\mathcal{CI}$-related metrics:

$\mathcal{CI}$_failure: a binary variable to encode the presence of build&test *failures* while running *Travis-CI*.

$\mathcal{CI}$_latency: the time interval in minutes from pull-request creation to the last commit tested by $\mathcal{CI}$. The team-wide code review typically starts after all commits have been tested.

3.3 Regression model

We used multiple linear regression to model the likelihood of pull-request acceptance and the latency of evaluation. Our models were fitted using the *lme4* [42] package in R.

In regression analysis, the slope of the regression equation would be exerted disproportionate impacts, which factitiously inflates the model's fit. Therefore, as customary in regression, we firstly removed outliers, e.g., very few pull-requests with an extremely large number of churns. Whenever one of our variables x was well fitted by an exponential distribution, we excluded as outlier values that exceeded $k(1 + 2/n)median(x) + \theta$ [43], where θ is the exponential parameter [44], n is the sample size, and k is computed such that not more than 2% of values are labeled as outliers. The summary of our metrics are shown in Table 2.

To study pull-request acceptance, we used the multi-level mixed effects logistic regression model because the outcome (i.e., PR_merged) is dichotomous and the dataset nested in two aspects (i.e., submitter and project). It means some pull-requests can come from the same projects or the same submitters. In our model, the metrics described in Subsection 3.2.2 were modeled as fixed effects; proj_id and submitter_id as random effects. For each fixed variable, we report its coefficients and significance level. According to the coefficients, we can calculate the odds ratio telling how much higher or lower are the odds that a pull-request would be accepted, by increasing per 'unit' of the measure. We consider coefficients important if they were statistically significant ($p < 0.05$). To evaluate the predictive power of logistic regressions, we present the *Area Under* the *receiver operating characteristic Curve* (AUC [45, 46]) as an assessment of the goodness of model fit for the logistic regressions.

Similarly, after outliers removal, we used the multiple linear mixed effects models to investigate the latency of pull-requests. In this paper, we only focused on the pull-requests which has been eventually merged, because: (1) if the pull-requests were rejected, the code changes would not affect the projects; (2) rejected pull-requests may be subject to different processes. We will further study rejected pull-requests in future work. In addition to the coefficients, we reported the effect size of each variable obtained from ANOVA analyses. The model's fit can be evaluated by pseudo *R-squared*s, i.e., the marginal (R_m^2) and conditional (R_c^2) coefficient of determination for generalized mixed-effects models [47, 48]. As implemented in the *MuMIn* package of R [49], (R_m^2) describes the proportion of variance explained by the fixed effects alone; and (R_c^2) describes the proportion of variance explained by both the fixed and random effects.

All numeric variables were first log transformed (plus 0.5 if necessary) to stabilize variance and reduce heteroscedasticity [50]. In addition, we computed the variance inflation factors (*VIF*s) for each predictor to check for multicollinearity. In our models, the *VIF*s of all remained factors were well below 3, indicating absence of multicollinearity [50]. There was no interaction between variables in the models, so it is easier for us to interpret our results and keeps the models clean. To study the effect of $\mathcal{CI}$-related variables on the pull-request process, we firstly builded a model using control predictors (i.e., without any $\mathcal{CI}$-related variables); then add $\mathcal{CI}$-related variables to generate the second model. In this way, we can know: (1) the overall impacts of $\mathcal{CI}$ features on the pull-requests process, reflected by the overall improvement of statistical model (i.e., the improvement of AUC and pseudo *R-squared*s after adding $\mathcal{CI}$-related variables); (2) the different effects of control predictors on the pull-request process with and without $\mathcal{CI}$ features.

4 Results

In this section, we separately discuss our models and results in terms of acceptance and latency.

4.1 Pull-request acceptance

Overall, both two models with and without $\mathcal{CI}$ related predictors (i.e., Model 1 and Model 2 in Table 3) can achieve remarkable performances, for the AUC (0.901 and 0.907) over the border of being a good model by certain standards (> 0.8) [51]. All the effects of predictors are consistent across two models, i.e., there is no effect flipping from positive to negative, vice versa. Therefore, we discuss their effects based on Model 2.

As for project-level metrics, the *proj_age* and *area_hotness* are statistically significant and positive for predicting acceptance. Perhaps unexpectedly, we find the odds of acceptance would increase by a factor of 1.296 ($\mathrm{e}^{0.259}$) for every e factor increase in project age, indicating that the more mature a project is, the more openness the project would be. This finding is different from the results presented in paper [16]. The area hotness is a strong predictor for acceptance. The likelihood of acceptance will increase by 2.014 ($\mathrm{e}^{0.700}$) times with per e factor change in *area_hotness*. We argue that there are more development tasks (e.g., coding, bug fixing or documenting), relating to the hot area in a project. Hence, the project may be eager to accept contributions trying to fulfil some relevant tasks. All pull-request level metrics are highly significant in the model. As expected, the large and complex pull-requests are less likely to be accepted, but submitting test cases with the pull-requests would moderately increase the probability of acceptance (1.298 times). For submitter-level metrics, we confirm the assumptions and findings in papers [16] that social factors play an importance role, very competitive with technical factors, in pull-based development process. Next, there are 4 workflow metrics that exert significant and negative effects on pull-request acceptance. The number of comments is the dominant one, and other predictors only

Table 3 Statistical models for pull-request acceptance

	Dependent variable:	
	(merged == TRUE)	
	Model 1	Model 2
log(proj_age)	0.377***	0.259***
log(team_size)	0.051	0.046
log(area_hotness + 0.5)	0.720***	0.700**
log(churn_additions + 0.5)	−0.112***	−0.123***
log(churn_deletions + 0.5)	0.059***	0.071***
log(n_commits)	0.319***	0.102***
log(description_complexity)	−0.107***	−0.116***
test_inclusion TRUE	0.245***	0.261***
core_team TRUE	0.761***	0.784***
social_strength	0.750***	0.679***
log(n_followers + 0.5)	0.138***	0.138***
log(n_comments + 0.5)	−0.448***	−0.565***
log(first_human_response + 0.5)	−0.015***	−0.028***
log(workload + 0.5)	−0.085**	−0.095***
log(integrator_availability + 0.5)	−0.050***	−0.040***
@mention_tag TRUE	0.001	0.019
#issue_tag TRUE	0.080**	0.060
Friday_effect TRUE	0.046	0.041
$\mathcal{CI}$_failure TRUE		−0.640***
log($\mathcal{CI}$_latency)		0.206***
Intercept	0.489	0.524
Akaike Inf. Crit.	31871.340	31030.380
Bayesian Inf. Crit.	32055.510	31232.090
AUC	0.901	0.907

Note: $^{*}p < 0.1$; $^{**}p < 0.05$; $^{***}p < 0.01$

have small effect sizes. With one log*(n_comments)* increased, the odds of acceptance will decrease by 43.2% (calculated as: $1 - \mathrm{e}^{-0.565}$) in our model. Thus, the more controversy in a pull-request, the more likely it is to be rejected. Lastly, we discuss the two $\mathcal{CI}$-related metrics. *$\mathcal{CI}$_failure* has a significant negative effect on acceptance as expected. The likelihood of rejection would greatly increase by 89.6% (calculated as: $1/\mathrm{e}^{-0.640}$), when the pull-requests have failed $\mathcal{CI}$ testing. However, if a pull-request go through a thorough $\mathcal{CI}$ test, the probability of acceptance will be raised, with the odds of 1.229 times for every unit of log*($\mathcal{CI}$_latency)* (measured the $\mathcal{CI}$ running time) increase.

Result: Most of traditional factors have expected effects on pull-request acceptance (RQ1). Surprisingly, we find project age is significant and positive for acceptance. The $\mathcal{CI}$ process indeed influence the acceptance of pull-request significantly (RQ2). If a pull-request cannot pass the $\mathcal{CI}$ checking stage, it would be 89.6% more likely to be rejected. For other metrics, there is no obviously change after $\mathcal{CI}$-related predictors included into the model (RQ3).

4.2 Pull-request latency

Model 1 has a relatively low goodness of fit ($R_m^2 = 38.9\%$, $R_c^2 = 50.5\%$). As expected, the pull-request churn, commit size, complexity, and length of discussion play a dominant role in explaining the variance in the data. All three effects are highly significant, and together account for 82.4% of the variance explained

Table 4 Statistical models for pull-request latency

	Dependent variable:			
	log(delta_t)			
	Model 1		Model 2	
	Coeffs.	Sum Sq.	Coeffs.	Sum Sq.
log(proj_age)	0.304***	788.8***	0.140***	886.6***
log(team_size)	−0.261***	1.3	−0.216***	12.7*
log(area_hotness + 0.5)	0.276	161.7***	0.370*	69.5*
log(churn_additions + 0.5)	0.149***	24697.2***	0.119***	25423.0***
log(churn_deletions + 0.5)	−0.017**	217.7***	−0.0001	140.6***
log(n_commits)	0.494***	16245.9**	−0.063***	16882.3***
log(description_complexity)	0.322***	4255.5***	0.279***	3752.8***
test_inclusion TRUE	0.146***	722.2***	0.070**	1188.1***
core_team TRUE	−0.571***	566.5***	−0.510***	510.1***
social_strength	0.031	621.7***	−0.025	536.5***
log(n_followers + 0.5)	−0.098***	201.2***	−0.081***	457.9***
log(n_comments + 0.5)	0.442***	23613.2***	0.127***	21369.7***
log(first_human_response + 0.5)	0.216***	9437.5***	0.195***	9481.5***
log(workload + 0.5)	0.250***	643.5***	0.241***	660.6***
log(integrator_availability + 0.5)	0.102***	886.1***	0.121***	1221.0***
@mention_tag TRUE	−0.109**	14.1*	−0.040	43.8***
#issue_tag TRUE	0.299***	439.6***	0.260***	311.0***
Friday_effect TRUE	0.195***	230.4***	0.179***	230.7***
$\mathcal{CI}$_failure TRUE			0.193***	162.1***
log($\mathcal{CI}$_latency)			0.432***	22500.5***
Intercept	2.912***		2.274***	
Akaike Inf. Crit.	159714.700		154068.200	
Bayesian Inf. Crit.	159902.000		154272.400	
R_m^2	0.389		0.471	
R_c^2	0.505		0.580	

Note: $^{*}p < 0.1$; $^{**}p < 0.05$; $^{***}p < 0.01$

(calculated as: the percentage of Sum Sq. of above variables accounting for the whole Sum Sq. of Model 1 in Table 4). The pull-requests with more discussion, consisting of more commits, and adding more lines of code are associated with longer evaluation latencies. Effects related to the submitter's track record and reputation are also highly significant, with smaller but still sizeable contributions to explaining the data variance. Pull-requests by the core team members and contributors with more followers are associated with shorter evaluation latencies. Perhaps more surprisingly, test case inclusion has highly significant positive effects, i.e., pull-requests with some test cases are associated with longer evaluation latencies. We present an assumption that the integrators may run the test code themselves for validation, and then merge the pull-request.

Model 2 offers a significantly better fit ($R_m^2 = 47.1\%$, $R_c^2 = 58.0\%$). Pull-request churn, commit size, complexity, and length of discussion, all highly significant, remain the most prominent predictors, together explaining 63.8% of the variance explained. However, the effect of commit size (*n_commits*) converts from positive to negative. It suggests that the patch separating into small pieces (i.e., commit your code frequently) will be processed quicker than a big patch, in the context of $\mathcal{CI}$, while holding other variables constant. This finding conforms to the theory of $\mathcal{CI}$ and agile development [52, 53]. For submitter-level metrics (social factors), the effect of *n_followers* increase more than twice. We consider that the integrators can trust the contribution from a reputed submitter more with the help of $\mathcal{CI}$. Except for *n_comments* which has been discussed above, the first human response and integrators' availability have sizeable effects, for the workflow metrics. Pull-requests with later initial reactions from integrators tend to also be closed later (9.0% of the variance explained). It suggests that the initial priorities integrators assign to pull-requests very early in the evaluation process (the median first comment time is 16 min) are already indicative of the (much later) closing time (median 11.2 h). Other variables have significant positive, albeit smaller, effects on pull-request latency, except for *@mention_tag*. Lastly, the CI-related factors are highly significant and cover more than 21.4% of the variance explained, on par with the main social and technical effects, i.e., pull-request size (*churn_additions*) and length of discussion (*n_comments*). The prominence of the total CI latency effect supports the process description in Subsection 2.1: integrators usually wait for the automatic testing phase to end (median: 39 min) before proceeding to do a team-wide code review and eventually close the pull-request.

Result: For RQ1, most of traditional factors influence the pull-request latency in an expected way, which is similar to the results of acceptance models. $\mathcal{CI}$ is a dominant factor of pull-request latency (RQ2), suggesting that integrators and code reviewers are willing to wait for the $\mathcal{CI}$ outcomes to assist them in making final decisions. A few predictors did change their effects after adding $\mathcal{CI}$-related variables (RQ3). Especially, the effect of commit size converts from positive to negative, which would indicate an agile principle that separating large code patch into small pieces and commit them more frequently.

5 Conclusion

Allowing greater inclusivity in contributions can result in a deluge of pull-requests, which, if unchecked, can significantly increase the burden on integrators in distributed software development projects. Our models for acceptance and latency show that pull-request evaluation is complex, and depends on many predictors. Naturally, a part of technical factors matter, e.g., the more succinct a pull-request is, the faster it will be reviewed, and the easier to be accepted. Software projects prefer to accept pull-requests with less controversy and originated from acquainted and trusted contributors. Other actionable strong predictors are the delay to the first human response and the availability of the CI pipeline. Improving on both may hasten the evaluation process.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61432020, 61472430, 61502512) and Postgraduate Innovation Fund of University of Defense Technology (Grant No. B130607). We thank Premkumar Devanbu, Vladimir Filkov and Bogdan Vasilescu for their very useful feedback on this paper.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Osterweil L. Software processes are software too. In: Proceedings of the 9th International Conference on Software Engineering. Los Alamitos: IEEE, 1987. 2–13

2 Jiang J J, Klein G, Hwang H-G, et al. An exploration of the relationship between software development process maturity and project performance. Inf Manag, 2004, 41: 279–288

3 Kogut B, Metiu A. Open-source software development and distributed innovation. Oxford Rev Econ Policy, 2001, 17: 248–264

4 Barr E T, Bird C, Rigby P C, et al. Cohesive and isolated development with branches. In: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering. Berlin/Heidelberg: Springer-Verlag, 2012. 316–331

5 Gousios G, Pinzger M, van Deursen A. An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering. New York: ACM, 2014. 345–355

6 Gousios G, Zaidman A, Storey M-A, et al. Work practices and challenges in pull-based development: the integrator's perspective. In: Proceedings of the 37th International Conference on Software Engineering. Piscataway: IEEE, 2015. 358–368

7 Bird C, Gourley A, Devanbu P, et al. Open borders? Immigration in open source projects. In: Proceedings of the 4th International Workshop on Mining Software Repositories. Washington, DC: IEEE, 2007. 6

8 Gharehyazie M, Posnett D, Vasilescu B, et al. Developer initiation and social interactions in OSS: a case study of the Apache Software Foundation. Empir Softw Eng, 2014, 20: 1318–1353

9 Gousios G, Storey M-A, Bacchelli A. Work practices and challenges in pull-based development: the contributor's perspective. In: Proceedings of the 38th International Conference on Software Engineering. New York: ACM, 2016. 285–296

10 Dabbish L, Stuart C, Tsay J, et al. 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: ACM, 2012. 1277–1286

11 Dabbish L, Stuart C, Tsay J, et al. Leveraging transparency. IEEE Softw, 2013, 30: 37–43

12 Yu Y, Wang H M, Yin G, et al. Who should review this pull-request: reviewer recommendation to expedite crowd collaboration. In: Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, Jeju, 2014. 335–342

13 Yu Y, Wang H M, Yin G, et al. Reviewer recommender of pull-requests in GitHub. In: Proceedings of the 2014 International Conference on Software Maintenance and Evolution. Washington, DC: IEEE, 2014. 609–612

14 Yu Y, Yin G, Wang H M, et al. Exploring the patterns of social behavior in GitHub. In: Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies. New York: ACM, 2014. 31–36

15 Pham R, Singer L, Liskin O, et al. Creating a shared understanding of testing culture on a social coding site. In: Proceedings of International Conference on Software Engineering. Piscataway: IEEE, 2013. 112–121

16 Tsay J, Dabbish L, Herbsleb J. Influence of social and technical factors for evaluating contribution in GitHub. In: Proceedings of the 36th International Conference on Software Engineering. New York: ACM, 2014. 356–366

17 Vasilescu B, Yu Y, Wang H M, et al. Quality and productivity outcomes relating to continuous integration in GitHub. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2015. 805–816

18 Yu Y, Wang H M, Filkov V, et al. Wait for it: determinants of pull request evaluation latency on GitHub. In: Proceedings of Working Conference on Mining Software Repositories, Florence, 2015. 367–371

19 Duvall P M, Matyas S, Glover A. Continuous Integration: Improving Software Quality and Reducing Risk. Boston: Pearson Education, 2007

20 Booch G. Object-Oriented Analysis and Design with Applications. 3rd ed. Redwood City: Addison Wesley Longman Publishing Co., Inc., 2004

21 Fowler M. Continuous integration, 2006. http://martinfowler.com/articles/continuousIntegration.html

22 Holck J, Jørgensen N. Continuous integration and quality assurance: a case study of two open source projects. Australas J Inform Syst, 2007, 11, doi: 10.3127/ajis.v11i1.145

23 Hars A, Ou S S. Working for free? Motivations of participating in open source projects. Int J Electron Comm, 2002, 6: 25–39

24 Dempsey B J, Weiss D, Jones P, et al. Who is an open source software developer? Commun ACM, 2002, 45: 67–72

25 Meyer M. Continuous integration and its tools. IEEE Softw, 2014, 31: 14–16

26 Vasilescu B, van Schuylenburg S, Wulms J, et al. Continuous integration in a social-coding world: empirical evidence from GitHub. In: Proceedings of International Conference on Software Maintenance and Evolution. New York: ACM, 2014. 401–405

27 Tsay J, Dabbish L, Herbsleb J. Let's talk about it: evaluating contributions through discussion in GitHub. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2014. 144–154

28 Hellendoorn V J, Devanbu P T, Bacchelli A. Will they like this? Evaluating code contributions with language models. In: Proceedings of Working Conference on Mining Software Repositories, Florence, 2015. 157–167

29 Hindle A, Barr E T, Su Z D, et al. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering. Piscataway: IEEE, 2012. 837–847

30 Nagappan N, Murphy B, Basili V. The influence of organizational structure on software quality: an empirical case study. In: Proceedings of the 30th International Conference on Software Engineering. New York: ACM, 2008. 521–530

31 Bettenburg N, Hassan A E. Studying the impact of social structures on software quality. In: Proceedings of the 18th International Conference on Program Comprehension, Braga, 2010. 124–133

32 Zimmermann T, Premraj R, Bettenburg N, et al. What makes a good bug report? IEEE Trans Softw Eng, 2010, 36: 618–643

33 Duc Anh N, Cruzes D S, Conradi R, et al. Empirical validation of human factors in predicting issue lead time in open source projects. In: Proceedings of International Conference on Predictive Models in Software Engineering. New York: ACM, 2011. 13

34 Vasilescu B, Filkov V, Serebrenik A. Perceptions of diversity on GitHub: a user survey. In: Proceedings of the 8th International Workshop on Cooperative and Human Aspects of Software Engineering. Piscataway: IEEE, 2015. 50–56

35 Gousios G. The GHTorrent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories. Piscataway: IEEE, 2013. 233–236

36 Vasilescu B, Posnett D, Ray B, et al. Gender and tenure diversity in GitHub teams. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems. New York: ACM, 2015. 3789–3798

37 Gousios G, Zaidman A. A dataset for pull-based development research. In: Proceedings of the 11th Working Conference

on Mining Software Repositories. New York: ACM, 2014. 368–371

38 Zhu J X, Zhou M H, Mockus A. Patterns of folder use and project popularity: a case study of GitHub repositories. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. New York: ACM, 2014. 30

39 Sauer C, Jeffery D R, Land L, et al. The effectiveness of software development technical reviews: a behaviorally motivated program of research. IEEE Trans Softw Eng, 2000, 26: 1–14

40 Rigby P C, Bird C. Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2013. 202–212

41 Śliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories. New York: ACM, 2005. 1–5

42 Bates D M. lme44: mixed-effects modeling with R. 2010. http://lme4.r-forge.r-project.org/lMMwR/lrgprt.pdf

43 Patel J K, Kapadia C H, Owen D B. Handbook of Statistical Distributions. New York: M. Dekker, 1976

44 Rousseeuw P J, Croux C. Alternatives to the median absolute deviation. J Amer Statist Assoc, 1993, 88: 1273–1283

45 Hanley J A, McNeil B J. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology, 1982, 143: 29–36

46 Robin X, Turck N, Hainard A, et al. pROC: an open-source package for R and S+ to analyze and compare ROC curves. BMC bioinform, 2011, 12: 77, doi: 10.1186/1471-2105-12-77

47 Johnson P C D. Extension of Nakagawa & Schielzeth's R2GLMM to random slopes models. Methods Ecol Evol, 2014, 5: 944–946

48 Nakagawa S, Schielzeth H. A general and simple method for obtaining R2 from generalized linear mixed-effects models. Methods Ecol Evol, 2013, 4: 133–142

49 Barton K, Barton M K. Package 'MuMIn'. 2015. https://cran.r-project.org/web/packages/MuMIn/MuMIn.pdf

50 Cohen J, Cohen P, West S G, et al. Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences. 3rd ed. New York: Routledge, 2013

51 Metz C E. Basic principles of ROC analysis. Semin Nucl Med, 1978, 8: 283–298

52 Stolberg S. Enabling agile testing through continuous integration. In: Proceedings of Agile Conference, Chicago, 2009. 369–374

53 Beck K. Embracing change with extreme programming. Computer, 1999, 32: 70–77