What the Fork: A Study of Inefficient and Efficient Forking Practices in Social Coding

Shurui Zhou
Carnegie Mellon University, USA

Bogdan Vasilescu
Carnegie Mellon University, USA

Christian Kästner
Carnegie Mellon University, USA

ABSTRACT

Forking and pull requests have been widely used in open-source communities as a uniform development and contribution mechanism, giving developers the flexibility to modify their own fork without affecting others before attempting to contribute back. However, not all projects use forks efficiently; many experience lost and duplicate contributions and fragmented communities. In this paper, we explore how open-source projects on GitHub differ with regard to forking inefficiencies. First, we observed that different communities experience these inefficiencies to widely different degrees and interviewed practitioners to understand why. Then, using multiple regression modeling, we analyzed which context factors correlate with fewer inefficiencies. We found that better modularity and centralized management are associated with more contributions and a higher fraction of accepted pull requests, suggesting specific best practices that project maintainers can adopt to reduce forking-related inefficiencies in their communities.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**; *Open source model.*

KEYWORDS

Fork-based development, Collaboration, Modularity, Centralized Management

ACM Reference Format:
Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2019. What the Fork: A Study of Inefficient and Efficient Forking Practices in Social Coding. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3338906.3338918

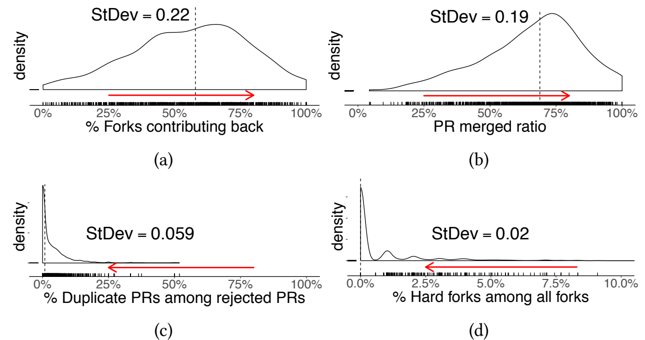


Figure 1: Density plots of four inefficient forking practices with high variance among projects in our sample. The arrow points towards higher efficiency. The dashed line shows the median.

1 INTRODUCTION

Collaboration is essential for software development at scale, in both industrial and open-source projects. Inadequate models of collaboration can stifle innovation and severely hurt common infrastructure, *e.g.*, when code structure does not align with team structure [43]. Although open source has achieved enormous productivity gains [25, 40], sustainability of the open-source movement is in question without models of collaboration that incorporate volunteers and avoid undue burden on few maintainers [28, 78]. In this paper we focus on a specific, important aspect: *how to effectively and efficiently collaborate on a common code base with forks.*

The notion of forking in open source has evolved: Traditionally, forking was the practice of copying a project and splitting off new independent development; forking was rare and was often intended to compete with or supersede the original project [34, 57, 59]. Nowadays, forks in distributed version control systems are public copies of repositories in which developers can make changes, potentially, but not necessarily, with the intention of integrating those changes back into the original repository. With the rise of social coding and explicit support in version control systems, forking of repositories has become very popular [38, 63]: *e.g.*, over 114,120 GitHub projects have more than 50 forks, and over 9,164 projects have more than 500 forks as of June 2019, with numbers rising quickly.

However, developing with forks does not come without costs. Independent development means contributions are not always visible to others, unless an explicit merge-back attempt is made. With a growing number of forks, coordination overhead rises [23]. Specifically, we note *four challenges:* (1) *lost contributions*—changes published in forks, but not integrated back into the original repository, are hard to find and de-facto lost for the larger community [84]; (2) *rejected pull requests*—developers who attempt to contribute but are rejected may be frustrated and refrain from future contributions [72]; (3) *redundant development*—developers may independently implement similar functionality in forks without coordination [38, 39, 84]; and (4) *fragmented communities*—a lower bar for forking can encourage developers to maintain multiple product variants in parallel, fragmenting the community and making it difficult for users to identify the variant of a project that best fits their needs or that is most actively developed [35]. Overall, we consider lost contributions, rejected pull requests, redundant development, and

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.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08...$15.00
https://doi.org/10.1145/3338906.3338918

fragmented communities as development process **inefficiencies** and we study how projects can reduce such inefficiencies.[1]

Prior work that studied the fork-based development model suggests high variance in how and why developers use forks. For example, Stănciulescu et al. [73] found for Marlin, a heavily-forked open-source firmware project for 3D printers (over 5,800 forks on GitHub), that most forks are inactive (no update after forking point) and used for configuration rather than new feature development, while only few are actively maintained. Similarly, Jiang et al. [45] found, after surveying 124 GitHub developers, that some created forks with the intention of contributing back (46 %), while others simply used them as a backup. We also found that GitHub projects are quite different with respect to the discussed inefficiencies, as shown in Figure 1 and as we will discuss in Section 3.

In this paper, we study the differences among open source communities in terms of forking practices, identify and measure inefficiencies, and model how characteristics and practices, such as modularity and centralized management, are associated with these inefficiencies. Specifically, we investigate the research question: **What characteristics and practices of a project associate with more efficient forking practices?** Understanding what influences inefficiencies can help communities to design interventions and to improve communication, onboarding, and sustainability.

Concretely, we *derived potential characteristics and practices* that could affect forking (in)efficiency by (a) asking open-source developers about their forking practices and (b) exploring exiting theories on distributed collaboration. We then tested these hypotheses at scale on GitHub data: We *designed measures* for four inefficiencies and potential characteristics and practices, collected data from 1131 GitHub projects with different number of forks, and used *multiple regression modeling*. We found that better modularity of the project structure and more centralized management practices for contributions are strong predictors of more contributions and more merged pull requests. Interestingly, our models also reveal a tradeoff: centralized management also associates with higher risk of community fragmentation through hard forks, as does a low pull request acceptance rate. Our results suggest best practices that project maintainers can adopt if they want to make fork-based development more efficient. Our operationalizations and results also lay the foundation for future tool support, such as benchmarking projects and highlighting inefficient practices [16].

In a nutshell, we make the following contributions:

- We design measures for open source community inefficiencies and observe differences among projects along these measures.
- We identify relevant context factors from literature analysis and 15 interviews and derive 8 corresponding hypotheses.
- We fit statistical models that associate context factors with inefficient practices across 1131 GitHub projects.

[1] Throughout this paper, we use the term *inefficiency* to concisely describe potential inefficiencies in collaboration. It is important to note that not all development that is not eventually integrated should be avoided. In fact, competition can be a driver for creativity or for exploring better solutions, and the drawbacks of redundancies might be outweighed by reduced coordination costs [12, 33, 52, 70, 80]. However, we argue, and find in our interviews, that modern forking practices often lead to inefficiencies that many developers would like to reduce. We aim to find ways in which project leads can influence practices (and their outcomes) if they chose to set such goals; it does not imply a goal to avoid all redundancies or always integrate all changes.

2 RELATED WORK: HISTORY OF FORKING

Before social coding, forking traditionally referred to the intention of splitting an independent development line (*e.g.*, forking *Jenkins* from *Hudson* over governance disagreements) to compete with the original repository, often under a new name. Here, we refer to **forking** in the sense of creating a public copy of a *git* repository, often with the goal of contributing to the original project; we refer to the traditional notion of splitting development as **hard forking**.

Hard forks have been discussed controversially: The right for hard forks (codified in open-source licenses) was seen as essential for guaranteeing freedom and useful for fostering disruptive innovations [34, 57, 59], encouraging a survival-of-the-fittest model [81], but hard forks themselves were often seen as antisocial and as risky to projects, since they could fragment a community and lead to confusion for both developers and users [34, 48, 57, 65]. There are not many cases where both communities survived after a hard fork, with a prominent exception being the BSD variants [63, 64, 68].

Past research on forking focused primarily on hard forks in open source, where a popular topic was understanding motivations for forking [21, 30, 48, 58, 68, 79]. For example, Nyman et al. [58] analyzed self-described reasons for hard forking, and found that variants targeting specific needs or user segments are the most common, followed by variants for different hardware (porting), bug fixes, and reviving abandoned projects. Researchers also argued that forking can be a suitable foundation for variant management [7, 30, 32, 71] and to overcome governance disputes [36].

Recent work focused on collaborative development with forks on social coding platforms. The openness of social coding creates transparency [23, 24] by making development activities in forks public and making pull request (PR) contributions visible. Prior work studied GitHub's pull-request model to investigate the reasons and factors that affect the PR evaluation process [38, 39, 77, 82]. Among the findings, both technical and social factors affect the chance of acceptance, such as the quality of the PR and the submitters' social connection to core members of the community.

Prior work confirmed also that forking provides increased opportunities for community engagement [23, 24, 38, 39, 53]; *e.g.*, over half of the commits in the Marlin project come from forks [73]. Biazzini et al. defined three collaboration models of open source projects on GitHub by understanding the dispersion of commits created by forks in the community, and revealed that collaboration patterns may differ significantly among projects [13]. More generally, it has been observed that communities often adopt a shared culture of common practices, but cultures can differ significantly among communities [15]. We study which aspects of a community's culture associates with collaboration efficiency.

Overall, most prior work focused on hard forks, though understanding the acceptance of contributions through individual pull requests has recently come in focus. In contrast, we study forks as a collaboration mechanism *at the project level* and focus on factors associated with project-wide inefficiencies.

3 INEFFICIENCIES IN FORKING

To motivate our research, we explore to what degree forking inefficiencies are common and differ across open-source projects.

Inefficiency: Lost contributions. Some developers publish changes in their own forks, but not also upstream, in the original project. Although technically public, these changes are hard to find and potentially lost for the larger community [84]. Fung et al. [35] report that only 14 % of all active forks of nine popular JavaScript GitHub projects integrated back any changes; extrapolating, this can amount to significant inefficiencies regarding development talent and lost effort across open source. In prior work [84], we found that developers are often interested in activities by other developers, but simply are not able to follow details in that many forks proactively—*e.g.*, they cite a developer discovering a feature in a fork as *"If it only exists in this fork, then I want to somehow get this feature into my fork."* Only very recently tools have been proposed to help developers monitor forks at scale [4, 66, 84].

We regard a community in which more developers *attempt to contribute* their changes upstream as more efficient. In our sample of 1131 GitHub projects, we identified the fraction of forks that attempt to contribute any changes back among all active forks (we will explain details regarding dataset and measurement in Section 4). In Fig 1(a), we show high variance across projects in the degree to which developers attempt to contribute their changes from forks back upstream, ranging from projects in which almost no forks attempt to contribute back, to projects where almost all forks do. These strong differences in observed efficiency raise the question of why these projects are so different and how project maintainers, if they wish to do so, can encourage more contribution attempts.

Inefficiency: Rejected pull requests. Not all attempted contributions are accepted by project maintainers. When developers submit a pull request (PR) that gets rejected, they can perceive this as a waste of their effort and get discouraged from contributing further [72]. One common reason for rejecting a PR is misalignment with the maintainers' vision of the project [8, 72]. From the community's perspective, a project in which most PRs are accepted can be considered as more effective with regard to contributor efforts. Observing the rate of rejected PRs among all closed PRs in our 1131 GitHub projects plotted in Fig 1(b) (details in Section 4), we see that in most projects a majority of PRs are accepted, but also note the high variance. Again, we would like to identify whether different project characteristics or practices can explain why some projects accept most PRs whereas others accept only a small percentage, and how project maintainers can strive for more efficiency.

Inefficiency: Redundant development. Globally decentralized fork-based development can be especially challenging because of fewer opportunities for rich interaction and direct communication [44]: Contributors can find it hard to maintain awareness of the project's trajectory or other developers [41, 42], which may lead to redundant development, in which two or more people independently work on similar functionality in forks without awareness of each other. Redundant development is a common reason for rejecting PRs in many projects [38, 72, 83]; in prior work, we and others have developed tooling to detect such duplicates [49, 67, 83]. Working on a change just to discover that other developers have previously or in parallel performed similar changes can also be demotivating. We consider projects that reduce accidental redundancies as more efficient.

Plotting the fraction of PRs rejected due to redundancies in Fig 1(c) (details in Section 4), we can observe that redundant development is a small but pervasive problem (mean 3.4 %; max 51 %). Again, we observe high variance across projects, worth investigating.

Inefficiency: Fragmented communities. Even on modern social coding platforms, diffusion of efforts, similar to those discussed for hard forks (Section 2), can be observed: Many secondary forks (*i.e.*, forks of forks) contribute to other forks, but not to the original repository; and forks slowly drift apart [35, 71]. Such fragmentation can threaten the sustainability of open-source projects when scarce resources are additionally scattered, causing redundant work. For example, *Marlin* has several hard forks, such as *Ultimaker* which has evolved into an independently managed project with over 170 own forks; fragmentation-related inefficiencies can be observed, for example, in a PR for Marlin fixing an issue (PR #10119) that was fixed already 2 years earlier in Ultimaker (PR #118). Hard forks are rare, but potentially expensive for a community. Plotting the percentage of hard forks among all the sampled forks of each project in Fig 1(d) (details in Section 4), we again observe high variance, raising the question of what kind of projects are more susceptible to hard forks.

In summary, we observe indicators of possible inefficiencies in most projects and high variance in how developers use forks. Suspecting that there are ways to actively encourage more efficient outcomes, we study to what degree project characteristics and practices associate with inefficiencies.

4 DETERMINANTS OF INEFFICIENCIES

We used a mixed-method approach to answer our research question: *What characteristics and practices of a project associate with more efficient forking practices?* We started qualitatively with interviews and literature analysis to identify candidate context factors (project characteristics and practices) that may influence effectiveness of fork-based development in a project, deriving *eight hypotheses.* We subsequently *operationalize measures* for context factors and inefficiencies, collect data from 1131 GitHub repositories, and quantitatively test our hypotheses using *multiple regression modeling*.

4.1 Identifying potential context factors

We pursued two strategies in parallel: interviews with active open-source contributors and analysis of the literature on distributed collaboration. This way, we collect perceptions of inefficiencies and their causes from practitioners and can contrast practices in different open-source systems, while at the same time also considering theories describing factors for efficient distributed collaboration, albeit established in contexts outside of fork-based development.

Specifically, we interviewed 15 maintainers and fork owners of several popular open-source projects, including *Bitcoin*, *Marlin*, *Smoothieware*, and *scikit-learn* (the number of forks ranged from 60 to 18.2K; all interviewees had public email addresses on their GitHub profiles), about efficient and inefficient practices and what might influence them. We stratified our sample of interviewees to include maintainers of projects with many forks, maintainers of projects with many duplicate PRs, developers who contributed to many open-source projects, and developers who made changes in

forks without attempting to contribute back. We conducted 12 interviews over Skype or email and 3 in person at two mixed academic-practitioner conferences. To analyze the transcripts, we conducted axial coding. This way we identified context factors that may affect the collaboration efficiencies, considering also the theories we found in the literature on distributed collaboration.

Modularity affects forking practices. *Interviews:* Discussions with contributors familiar with both *Marlin* and *Smoothieware* revealed an interesting contrast: *Marlin* (6,600 forks) and *Smoothieware* (720 forks) are both frequently forked open-source 3D printer firmware projects, but contributors familiar with both perceive very different practices. Learning from *Marlin*'s maintainability challenges due to crosscutting implementations, *Smoothieware* was designed modularly and emphasizes loose coupling and extension through separate modules, so that developers can add functionality without having to modify *Smoothieware*'s core. A developer who is familiar with both projects indicated that *Smoothieware* follows more professional and industrial development practices, such as submitting smaller and more cohesive changes. One interviewee indicated not contributing their significant extensions back to *Marlin* because of high integration effort. On GitHub, some projects have an extremely modular structure, *e.g.*, *homebrew* contains a collection of scripts or plug-ins that are assembled automatically, such that many contributions simply add files instead of modifying existing ones.

Modularity was not entirely uncontroversial in our interviews though, e.g., one *Smoothieware* contributor suggested that modularity helped with some extensions, but made others harder: *"So many restrictions that you can't just modify anything in the base code. [...] All this makes the code upgradeable, clean, and manageable, but the development progress is much slower because [...] some functions cannot be integrated with those restrictions."* This suggests tradeoffs regarding the rigidity that modularity imposes on developers, making certain changes hard or impossible.

Literature: Our interview observations align with theory (outside of social forking contexts) about the importance of modularity for (distributed) collaboration. For example, Parnas [61] and Conway [22] have both argued for the importance of modularity for collaboration and division of labor. Herbsleb and Grinter [43] found modularity, aligning with work assignments, to be essential for a geographically distributed software project. Researchers and practitioners have also emphasized the importance of modularity for open-source development [47, 55, 75]. For example, Torvalds [75] claims *"for without [modularity], you cannot have people working in parallel"* and Midha and Palvia [55] found that modularity is positively associated to technical success of open-source projects. Specifically, MacCormack et al. [51] suggested that more modular projects could be more attractive to potential contributors. It is hence plausible that modularity also has positive effects on collaboration efficiency in fork-based development among loosely-connected developers on social coding platforms. At the same time, researchers have found that many aspects of a software system are difficult to implement modularity [74] and that too rigid compatibility requirements might hinder innovation [15]; also, modularity does not always align with how developers think [60].

Hypotheses: Despite raised concerns, we hypothesize that a modular design of the software would make it easier to contribute to a project, which influences both whether developers attempt to contribute and to what degree maintainers accept contributions:

$\mathbf{H}_1$. *Projects with a better modular design have a larger fraction of contributing forks.*

$\mathbf{H}_2$. *Projects with a better modular design have a larger fraction of merged pull requests.*

Coordination mechanisms affect forking practices. *Interviews:* Interviewees of many projects, including *Marlin* and *Smoothieware*, indicated that their communities welcome all PRs that may benefit the larger community and that they are interested in activities in various forks, though they find it hard to monitor them. In contrast, an interviewee from the cryptocurrency project *Bitcoin* (22,100 forks) expressed a different view: *Bitcoin* has adopted a central management style, in which a relatively stable team of core developers decides the direction of the project, and in which features are discussed and decided upfront in an issue tracker (often political and hard fought among different camps [50]). The issue tracker records which forks contain the corresponding code changes for each issue; other forks are of little interest to maintainers and unsolicited PRs remain ignored for years. Similarly, one of the maintainers of the Python machine-learning project *scikit-learn* (16,300 forks) indicated that developers have little chance of integrating their changes upstream unless they talk to the maintainers first.

Developers also perceive explicit coordination as a key mechanism to avoid redundant development. Certain open-source communities perceived redundancies as a significant problem and promoted explicit coordination to combat it; *e.g.*, *Django* adopted a policy requiring contributors to communicate with the core team upfront to *claim* issues before submitting patches [1]. A maintainer of *scikit-learn* was even surprised about the existence of duplicate PRs, because in their project explicit coordination (developers discuss with the core team before doing any work) is the norm.

Literature: Researchers have long studied different degrees of explicit coordination and their tradeoffs in distributed collaboration, often in corporate settings, e.g., Brandts and Cooper [17] found that central coordination makes it easier to manage a division's product types but more difficult to take advantage of each division's private information. Comparing Linux and Wikipedia to traditional organizations, Puranam et al. [62] observed that Linux uses a *centralized task-division strategy* in which the initial problem formulation is defined by the founder of a project, while Wikipedia's task division is decentralized, which the researchers associate with problems of misinformation and duplication contributions. Regarding *task allocation*, Linux and Wikipedia are both decentralized, so that tasks are allocated through voluntary, self-selection of members into roles. Shaikh and Henfridsson [69] studied the version control history of Linux and observed that Linux changed its management strategies as the community evolved—from centralized to decentralized: The authors argued that the governance strategy is a configuration of coordination processes, and governance varies across open source communities. This matches our observation of different communities with different coordination strategies. We expect to see similar tradeoffs among coordination strategies also in new forms of collaboration with forks in open source.

Hypotheses: We hypothesize that projects coordinating contributions upfront in an issue tracker reduce inefficiencies by encouraging more focused development activities that are more frequently integrated, and rejecting fewer PRs because fewer PRs misalign with the maintainer's vision. We also hypothesize that *pre-communication*, *i.e.*, developers discussing their contributions before submitting PRs, associates with fewer redundant PRs:

$\mathbf{H}_3$. *Projects pursuing a centralized management strategy have a larger fraction of contributing forks.*

$\mathbf{H}_4$. *Projects pursuing a centralized management strategy have a larger fraction of merged pull requests.*

$\mathbf{H}_5$. *Projects in which external developers tend to discuss or claim an issue before submitting pull requests have a lower frequency of redundant development.*

Contribution barriers affect community fragmentation. *Interviews:* Some interviewees indicated that contribution barriers led them to create a hard fork, *e.g.*, the owner of a video recording project explained *"I submitted a PR but they rejected it. Because it is incompatible to the maintainer's vision [...] so I think, fine, I will keep my own fork."* Later, this fork started to attract its own external contributions. Also, as one Smoothieware interviewee said (quote above), the rigidity that modularity imposes on developers makes integrating certain changes hard or impossible, leading in some cases to active but unmerged development; Bitcoin, with its rigorous centralized management, is one of the projects that has the most hard forks. Disagreements between maintainers and contributors can lead to hard forks and fragment communities.

Literature: As discussed in Section 2, reasons for hard forks have been well studied (before the rise of social coding and distributed version control), and conflicts between the project leader's vision and the needs of community members were a common cause [8, 56].

Hypotheses: We hypothesize that a low rate of accepted external contributions, modularity restrictions, and centralized management all can trigger community fragmentation:

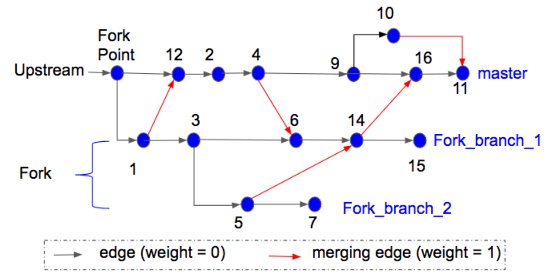
$\mathbf{H}_6$. *Projects with a lower pull request merge ratio have higher likelihood of having at least one hard fork.*

$\mathbf{H}_7$. *Projects with a more modular design have higher likelihood of having at least one hard fork.*

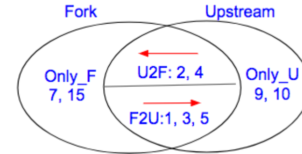
$\mathbf{H}_8$. *Projects pursuing a centralized management strategy have higher likelihood of having at least one hard fork.*

Note that $\mathbf{H}_6$ uses the PR merge ratio (the predicted outcome in $\mathbf{H}_2$ and $\mathbf{H}_4$) as a factor. That is, we expect potential *tradeoffs*, in that factors that improve efficiency regarding merged PRs could at the same time reduce efficiency regarding community fragmentation.

Summary and importance. Modularity and coordination are established theories in software engineering. After reviewing the literature and interviewing open source contributors, we derived eight hypotheses about context factors informed by the two theories, that are expected to associate with inefficient outcomes in a domain where the theories have not been tested before fork-based development. To test these hypotheses, we operationalize our context factors in GitHub trace data and model their effects at scale across many open source projects (details in Section 4.2). This way, we not only test the limits of the two theories and expand them in the new domain of fork-based development, but also provide quantitative empirical evidence on the effects of the different context factors on relevant outcomes, where previously there were only beliefs. This step of providing data-driven empirical evidence to popular theories is particularly important, as beliefs and evidence often misalign in software engineering practice [11, 26].



(a) Commit history of fork and upstream



(b) Only_F: only exist in fork; Only_U: only exist in upstream; F2U: merged from fork to upstream; U2F: pulled from upstream to fork.

Figure 2: Determining the origin of commits.

4.2 Operationalizations

We iteratively developed outcome measures for inefficiencies, measures for context factors, and measures for control variables. Specifically, we first developed an initial measure and subsequently validated construct validity by manually checking samples and outliers, repeating the process with a refined measure as needed. Several measures are nontrivial and are built on top of significant prior research, as we will discuss. We share implementations for all measurements as part of our replication package [6].

Outcome: Ratio of contributing forks. To assess inefficiencies regarding lost contributions, we measure the fraction of active forks in which developers have submitted PRs or otherwise integrated their code changes into the upstream project (higher values indicate higher efficiency). Specifically, we query the GitHub API to identify whether PRs have been issued for any commits from a fork. We also analyze the commit histories to identify whether commits have been merged without publicly visible PRs.

Unfortunately, reliably detecting active forks and merged changes is not trivial. Forks may pull changes from upstream, upstream repositories can merge changes also without PRs, commits are often merged across various branches, and commit timestamps are not generally reliable. Hence, we developed a new approach to identify from which fork a commit originates and how it has been merged across branches and forks.

To this end, we analyze the joint commit graph of the fork and the upstream repository (nodes are commits, edges are parent relationships, merge commits have multiple parents). Since commits may be merged multiple times and in different directions across branches and forks, we analyze the number of merge commits and assign a commit as originating in the fork from which *it was merged*

the fewest times, as follows:

- Each branch in the fork and the upstream repository corresponds to a commit node in the graph (usually a node without children). For merge commits, we distinguish between the direct parent (first parent) and the merged parents (other parents) of a commit.
- We assign a weight of 1 to an edge between a merge commit and its merge parents and a weight of 0 to all other edges.
- For every commit node, the shortest path from that node to a commit node mapped to a branch indicates the branch and thus the repository the commit originates from.
- If there is no path from a commit to any branch of a repository, it has not been merged into that repository yet.

We illustrate an example in Figure 2(a): Commit node 5 has been merged from a branch into another branch and from the fork into the upstream master; by counting the merge edges, we can identify that it originates from the fork because more merge edges need to be traversed to reach a branch from the upstream repository; similarly, we can identify that commit node 2 originates from upstream; there is no path from commit node 7 to the upstream repository, indicating that the commit originates from the fork and has not been merged yet. Note, a similar mechanism to recognize the origin of commits was suggested in prior work [13], but without a description of how to perform it and without releasing an implementation.

To measure the ratio of contributing forks, we determine which forks are active (i.e., have commits originating from the fork), then identify successful and attempted contributions from merged commits in the commit graph and from PRs originating from the fork.

Outcome: Ratio of merged PRs. To assess inefficiencies regarding rejected PRs, we measure the fraction of closed PRs that have been accepted (higher values indicate more efficient outcomes). The resolution status reported by GitHub is often not reliable [38], as many developers integrate PRs through other mechanisms than GitHub's user interface, thus closing them without marking them as accepted. We follow Gousios' heuristics [38] to identify accepted contributions, but refine them to account for frequent practices we observed:

- If the PR is marked as merged on GitHub, we mark it as accepted. (83.2% of all merged PRs).
- If a commit closes the PR (using certain phrase conventions advocated by GitHub, *e.g.*, `fixes #1234`) and that commit appears in the target project's branch, we consider the PR as accepted. Different from Gousios' work, we use GitHub's issue events timeline API, rather than analyzing textual comments, to detect links to PRs in commit messages. (8.8% of all merged PRs).
- If any of the last 3 discussion comments of the PR refers to a commit SHA, we consider the PR as accepted. Specifically, we follow Gousios' criteria: (1) the comment contains a reference to a specific commit identifier (SHA), (2) this commit SHA appears in the project's master branch, and (3) the comment can be matched by the regular expression `(merg|apply|appl|pull|push|integrat|land|cherry(-|\s+)pick|squash)(ing|i?ed)`. We extended this by making sure that no second linked PR appears in the comment, indicating a competing or superseding PR. (0.15% of all merged PRs).
- If the last comment before closing the PR matches both rules (1) and (2) above, or matches only rule (3), we consider the PR as accepted, unless a link to another PR appears in the comment. (7.9% of all merged PRs).

If no heuristic identifies a PR as accepted, we mark it as rejected.

Outcome: Ratio of duplicate PRs. To assess inefficiencies regarding duplicate development, we measure the fraction of closed PRs rejected due to redundant work (lower values indicate higher efficiency). To identify duplicate PRs, we refined heuristics, summarized and validated by Yu et al. [83], based on regular expressions to identify duplicate-related keywords in PR comments and links to other PRs. We also found many cases in which a PR is redundant to a commit so we extend the link detection to include commit SHAs. After several rounds of refinement, we arrived at six patterns for detecting PRs rejected due to redundant development that can be found in the implementation [6].

Outcome: Presence of hard forks. To assess inefficiencies regarding community fragmentation, we measure whether projects have at least one hard fork. We consider a fork as a hard fork if (a) it has attracted its own external contributions (at least two PRs submitted by other contributors) or (b) it has substantial unmerged changes (at least 100 commits, as identified from our commit graph, see Figure 2) and the project's name has been changed (with Levenshtein distance > 2). In our sample, 28 % of the projects have at least one hard fork, as per our operationalization.

Predictor for modularity: Logic coupling index. Researchers have proposed different metrics to measure the modularity of a project, taking different perspectives. For example, many approaches use program analysis to detect dependencies among program structures [14, 29]; others measure logic coupling from co-change patterns observed in the project's revision history [10, 18, 85]. To measure modularity uniformly across projects in different programming languages, we adopt a light-weight previously validated measurement of logic coupling, *ROSE* [85]: We define the *logic coupling index* of a commit as the fraction of file pairs that have been changed together in that commit out of all file pairs in the project. We aggregate this measure at the project level by computing the median of recent commits. To focus on modularity relevant to external contributors and avoid bias from past but now changed practices, for each project, we analyze the last 50 commits whose authors are external contributors (the results are robust for different operationalizations with the last 100 or 500 commits). A lower logic coupling index indicates better modularity, as fewer files are changed together.

Predictor for modularity: Additive contribution index. In addition to logic coupling, we also measure the modularity of contributions in terms of whether they add or modify code. This measure is motivated by observations, discussed above, that some GitHub projects have an extreme form of modularity in that they primarily collect extensions or plug-ins and are extended by contributing additional files rather than editing existing ones. Thus, we define a second modularity measure, the *additive contribution index*, that measures to what degree external contributions are additive: We measure the fraction of new files added out of all files touched per commits. We compute the median over results of all commits from external contributors in a project. A higher additive contribution index indicates that more changes were additive in nature, indicating better modularity from a contributors perspective.

Predictor for coordination: Centralized management index. We measure the degree developers use the issue tracker to

coordinate *what* to work on *before* submitting a pull request: We observe which new pull requests are linked to existing issues (typically by referring to the issue number in the text of the pull request) by parsing the event timeline of the pull request provided by the GitHub API. We define the *centralized management index* of a project as the fraction of pull requests that link to issues out of all closed pull requests from external contributors. A higher centralized management index indicates that upfront coordination on *what* to work on through issues is more common in a project.

Predictor for coordination: Pre-communication index. We additionally measure to what degree developers coordinate *who* will work on an issue *before* submitting a pull request by observing whether developers 'claim' an issue before completing the work. Specifically, we look for two commonly recommended practices of *pre-communication* before submitting a final PR: (1) Developers might leave a comment on the issue to which they later respond, indicating their plan to work on the issue and possibly linking to their fork. (2) Following explicit recommendations from GitHub,[2] developers might submit an incomplete PR clearly marked as 'work in progress' (*e.g.*, using labels) and later update that PR once they finish their work. Both practices publicly announce that a developer is working on an issue. We define the *pre-communcation index* of a project as the fraction of PRs for which the author has commented under the linked issue *before* submitting the PR or in which the PR was marked as work in progress in its history out of all closed PRs by external contributors that are linked to issues. A higher pre-communcation index indicates that the practice of coordinating *who* will work on an issue is more common in a project.

Control variables. Finally, we measure a number of controls that might co-vary with our efficiency outcomes. Specifically, we collect form the GitHub API the project age, size (in bytes), and number of forks – older, bigger, or more heavily forked projects are likely to adopt different practices. We additionally collect project-level aggregate statistics about all closed PRs by *external* (non-core) contributors, modeled closely after factors that prior research found to correlate with the chance of accepting individual PRs [38, 77]: (1) SubmitterPriorExperience – a dummy encoding whether at least half of the PRs in the project are submitted by people with prior experience submitting and having merged PRs in the same project in the past; PRs from people with prior experience are more likely to be accepted [38]. (2) RatioPRsWithTests – the ratio of PRs containing test cases; PRs containing test cases are more likely to be accepted [38]. We reused our measure to identify tests [76] based on file name patterns maintained by the package search service npms.io, such as matching file paths containing *test* or *spec*. (3) PRHotness – the median over PRs of the number of commits on files touched by each PR during the previous three months prior to the PR creation; PRs touching "hot" files, changed frequently in the recent past, are more likely to be accepted [38]. and (4) SubmitterSocialConnections – a dummy encoding whether at least half of the PRs in the project are submitted by people who followed (already at PR creation time) the maintainer who closed each respective PR; PRs by more socially connected submitters, who follow the maintainers, are more likely to be accepted [77].

[2] https://blog.github.com/2015-01-21-how-to-write-the-perfect-pull-request/

Table 1: How we stratified our sample.

Group	Num. forks	Num. projects on GitHub	Num. projects in sample
A	[3,000 , +]	231	200
B	[1,000 , 3,000)	847	300
C	[20 , 1,000)	116,532	1300

4.3 Data Collection

We assembled a multidimensional dataset of actively-developed GitHub open-source projects with at least a moderate number of forks. Starting from a list of 137,424 projects with at least 20 forks in the March 2018 GHTorrent [37] dump, we filtered projects based on the following criteria:

- *Projects should be developing software applications or frameworks.* Interested in understanding software-development practices, we remove projects using GitHub for document storage or course project submission. We search for keywords like 'homework', 'assignments', 'course' to find online courses, remove projects starting with 'awesome-' (usually document collections), and remove projects with no programming-language-specific files.
- *Projects should have at least 10 commits, 10 active forks, and 1 closed pull request.* We are interested in active projects with some development history and some collaboration, so we set a minimum threshold of 10 commits, 10 *active* forks (*i.e.*, those with at least one own commit after forking), and at least one pull request by an external contributor.
- *Projects should have at least one closed issue.* Finally, we exclude projects that do not use the issue tracker, because we cannot establish coordination practices for those.

To not bias our analysis by practices applied by the largest or by many small projects, we stratify across projects with different numbers of forks, sampling 200 very frequently forked projects, 300 frequently forked, and 1300 moderately forked, as shown in Table 1; in each stratum we select a random sample. Finally, we exclude all projects from which we have previously interviewed developers and duplicate projects, resulting in 1131 projects for our analysis.

For each project, we need to analyze forks, external commits, external PRs, and issues. We only consider external PRs and external commits by developers who are not project owners and have not closed PRs of others in the project.

Since computing the flow graph in Fig. 2 requires locally cloning all forks in a project and is computationally expensive, we sample 100 active forks per project, that were forked more than 30 days before our analysis, to allow for time for developers to attempt to contribute changes back. We use the GitHub API to fetch the history of each issue and links among issues and PRs.

In Figure 3, we show the ranges and distributions of the four operationalized measures of modularity and coordination in our dataset. Note the large variance across projects for all variables.

4.4 Statistical Analysis

We use multiple regression modeling to test, for each outcome, whether it is significantly associated with the different hypothesized context factors, while controling for known confounding variables, cf. prior work. The multivariate nature of our analysis is especially relevant when modeling the PR merge ratio, which is known to be impacted by the presence of tests and the prior experience of the PR submitters [38, 77].

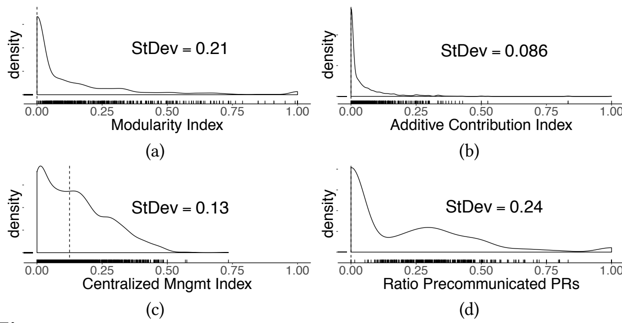


Figure 3: Density plots for our main predictors. The dashed line denotes the median.

Note that we perform our analysis *at project level* (each row of data aggregates information about one project), *i.e.*, we compare how projects with different characteristics and practices tend to differ regarding forking inefficiencies, on average.

For the *binary* outcome variable (presence of hard forks), we build a standard logistic regression model. Notably, we also build logistic regression models for the other three *ratio* outcome variables. Logistic regression is more appropriate when trying to estimate probabilities of frequencies (ratios) than linear regression, because in the latter case the binomial probabilities would become increasingly spiked as the number of observations increases; *e.g.*, the case with 50 pull requests merged out of 100 submitted gives more information than the case with 1 merged out of 2 submitted. In a GLM, the denominator from the ratio (*e.g.*, 100 for the former example and 2 for the latter) can be specified explicitly as the *weights* parameter when using the *glm* function in R.

When building the regression models, we take several steps to ensure robustness and validity. First, we conservatively remove the top up to 1 % of the data for variables with exponential distributions; these outliers tend to have high leverage, decreasing the models' robustness. We also test for high-leverage points using Cook's distance measure, and exclude additional projects from each model as needed; below each regression model summary table in Section 5 we show the exact number of data points modeled. Second, we test and correct for multicollinearity using the variance inflation factor (VIF). Third, we evaluate the goodness-of-fit of our models using McFadden's pseudo-R^2 measure. Finally, we report, for each model variable, its exponentiated coefficient (*i.e.*, its odds ratio – the factor by which a one unit increase in a predictor increases – if greater than 1 – or decreases – if less than 1 – the odds of the outcome occurring), standard error, significance level (p-value), and effect size (*i.e.*, η^2 – the fraction of deviance explained by the model that can be attributed to that predictor, as per an ANOVA type-II analysis; see columns "LR Chisq" in the model tables for the absolute amounts of deviance explained).

4.5 Threats to Validity

As usual, our operationalized measures can only capture some aspect of the underlying quality. For example, logic coupling at the file level may miss some more granular dependencies that may make changes challenging and our centralized-management index may miss rare practices such as coordinating in a separate channel.

As discussed, we manually validated construct validity of each measure on a sample of projects to avoid systematic errors and explored different operationalizations to ensure robustness. While we cannot exclude some noise, regression across over one thousand projects will likely pick up on signals despite some noise in measurements. Nonetheless, our results must be interpreted in the context of our operationalization. To this end, we share an R notebook detailing our analysis [6].

Finally, one must be careful to generalize our results beyond the context of our analysis of social coding in open-source projects on GitHub. Although many companies increasingly adopt practices from open-source development [46], they likely do not share the same context of loosely-coordinated distributed contributions from developers outside a core team.

Table 2: Contributing forks model (R^2 = 17%).

	Ratio contributing forks	
	Coeffs (Errors)	LR Chisq
(Intercept)	0.94 (0.05)	
NumForks	0.78 (0.01)***	2631.77***
Size	1.14 (0.00)***	1109.29***
ProjectAge	1.00 (0.00)***	147.27***
CentralizedMngmtIndex	6.03 (0.06)***	868.03***
ModularityIndex	1.23 (0.03)***	35.72***
AdditiveContributionIndex	0.97 (0.11)	0.09

***$p < 0.001$, **$p < 0.01$, *$p < 0.05$ N = 1131

5 RESULTS

In the following, we discuss results from hypothesis testing organized by forking inefficiency (outcomes).

When do forks attempt to contribute back? (H_1, H_3) To test our hypotheses that modularity (H_1) and coordination practices (H_3) associate with higher rates of attempted contributions, we modeled a project's *ratio of contributing forks* as a function of the two modularity indices and the centralized management index, while controlling for the overall number of forks, the project size, and the project age.

In Table 2, we show a summary of the regression model. Interpreting the coefficients, we first note a strong positive effect for the centralized management index, explaining approximately 18 % of the deviance explained by the model: projects with stronger coordination practices, as evidenced by advanced planning of what work needs to be done through issue linking, tend to have a higher fraction of contributing forks that submit patches upstream. Modularity in terms of logic coupling also has a positive effect, albeit weaker, accounting for about 1 % of the deviance explained by the model: projects with more modular architecture, in which changes can be made in relative isolation, without touching many files, tend to have a higher fraction of contributing forks. Therefore, we find evidence in support of both H_1 and H_3.

When are more contributions integrated? (H_2, H_4) To test our hypotheses whether modularity (H_2) and coordination mechanisms (H_4) may also facilitate the integration of changes originating in forks back into the upstream project, we modeled the *ratio of merged pull requests* submitted by external contributors, as a function of the modularity and centralized management indices. In the regression we control for known confounding factors, as per prior work: the total number of forks, the project size and age, the prior

Table 3: External PR merge ratio model (R^2 = 27%).

	Ratio merged PRs	
	Coeffs (Errors)	LR Chisq
(Intercept)	2.82 (0.04)***	
NumForks	0.82 (0.00)***	3001.50***
Size	1.08 (0.00)***	862.77***
ProjectAge	1.00 (0.00)***	355.78***
SubmitterPriorExperienceTRUE	1.33 (0.01)***	1084.06***
SubmitterSocialConnectionsTRUE	1.10 (0.01)***	124.74***
PRHotness	1.01 (0.01)*	5.99*
RatioPRsWithTests	1.35 (0.06)***	23.07***
CentralizedMngmtIndex	1.67 (0.03)***	226.64***
ModularityIndex	1.50 (0.02)***	308.46***
AdditiveContributionIndex	1.47 (0.07)***	30.28***
***$p < 0.001$, **$p < 0.01$, *$p < 0.05$	N = 1125	

Table 4: Duplicate PR ratio model (R^2 = 4%).

	Ratio duplicate PRs	
	Coeffs (Errors)	LR Chisq
(Intercept)	0.01 (0.09)***	
NumForks	1.16 (0.01)***	245.03***
Size	0.97 (0.01)***	19.03***
ProjectAge	1.00 (0.00)***	29.45***
RatioPrecommunicatedPRs	0.84 (0.06)**	7.81**
***$p < 0.001$, **$p < 0.01$, *$p < 0.05$	N = 1127	

experience of the pull request submitters, the ratio of pull requests containing test cases, and the median PR hotness.

In Table 3, we summarize the regression results. As expected, most (90 %) of the deviance explained by the model is attributed to the control variables. Still, even after controlling for confounds, all three main predictors have sizeable, positive effects on the average pull request merge ratio. Modularity, operationalized as low logical coupling and high ratio of added files to modified files, has the strongest effect (6 % of the deviance explained for the two variables together): the more modular the architecture, the higher the fraction of merged pull requests. Coordination also has a positive and comparably large effect (4 % of the deviance explained): the more planned the pull requests are, i.e., in response to open issues, the higher the average acceptance rate, other variables held constant. Together, these results provide strong support for both $\mathbf{H}_2$ and $\mathbf{H}_4$.

When is duplicate work more common? ($\mathbf{H}_5$) To test whether discussing or claiming an issue before submitting a PR correlates with less redundant development ($\mathbf{H}_5$), we modeled the average rate of duplicate pull requests per project, as a function of the rate of pre-communicated pull requests, controlling for project age, project size, and number of forks (older projects and bigger projects, with more forks, can be expected to experience more duplication, on average).

The regression summary in Table 4 suggests that the higher the rate at which pull requests are pre-communicated, the lower the overall rate of duplication among pull requests. However, we model rare events (both duplicates and pre-communication are relatively rare in our dataset), the model fit is rather poor (R^2 = 4 %), and our pre-communication index explains only 3 % of the deviance explained by the model. We conclude cautiously that: there is only weak evidence that claiming pull requests before working on them associates with lower risk of duplicate work.

When does the community risk fragmentation? ($\mathbf{H}_6$–$\mathbf{H}_8$) To test whether projects that reject many external contributions ($\mathbf{H}_6$), have a more modular design ($\mathbf{H}_7$), or have higher coordination requirements ($\mathbf{H}_8$), correlate with fragmented communities and hard forks, we modeled the likelihood of a project having hard forks as a function of the average external pull request merge ratio and the modularity and centralized management indices, while controlling for project size and the overall number of forks.

Table 5: Hard forks model (R^2 = 10%).

	Has hard forks (T/F)	
	Coeffs (Errors)	LR Chisq
(Intercept)	0.19 (0.49)***	
NumForks	1.25 (0.05)***	23.74***
Size	1.09 (0.04)*	5.76*
CentralizedMngmtIndex	4.92 (0.58)**	7.39**
ModularityIndex	0.66 (0.32)	1.57
AdditiveContributionIndex	4.32 (0.93)	2.43
PRMergeRatio	0.14 (0.42)***	22.24***
***$p < 0.001$, **$p < 0.01$, *$p < 0.05$	N = 1131	

Our model, summarized in Table 5, confirms a sizeable negative effect for the pull request merge ratio (35 % of the deviance explained), strongly supporting $\mathbf{H}_6$: the lower the pull request acceptance rate, the higher the chance of a project having hard forks, on average. The centralized management index also has a statistically significant positive effect (12 % of the deviance explained), supporting $\mathbf{H}_8$: more coordination requirements are associated with a higher risk of community members fragmenting into various hard forks. We do not find a statistically significant effect though for the modularity associating with hard forks ($\mathbf{H}_7$).

5.1 Discussion

Modularity. Modularity has been widely recognized as an important quality that facilitates software evolution and eases division of labor and collaboration [9, 22, 54, 61]. Our study confirms that better modularity is associated with higher efficiency of distributed fork-based development, specifically higher fraction of developers contributing their changes back ($\mathbf{H}_1$) and higher rate of integration of external contributions ($\mathbf{H}_2$). Note that logic coupling was beneficial in general, whereas extreme modularity where contributions are mostly additive do not seem to encourage a higher percentage of developers to contribute back but it significantly eases integration.

While there are some concerns about limiting effects of modularity for certain changes, even to the extent we could hypothesize potential fragmentation of communities through hard forks, we did not find in our models any *direct* evidence supporting these concerns ($\mathbf{H}_7$). However, there is a noteworthy *indirect* effect: higher modularity is associated with higher PR acceptance ratios ($\mathbf{H}_2$); in turn, higher PR acceptance ratios are associated with higher likelihood of community fragmentation through hard forks ($\mathbf{H}_6$). More research is needed to disentangle the effects of modularity more precisely from those of lower PR acceptance rates; we suggest this as a promising direction for future research.

In short, our results suggest a net-positive impact of modularity in fork-based collaborative development, a new domain lacking the empirical evidence.

Coordination. Our study also indicates the importance of active coordination among developers. Even though fork-based development on a transparent platform allows all developers to freely fork

projects, make changes without coordination, and suggest pull requests once done [23], coordination is associated with significant improvements to the efficiency of a community regarding forking outcomes specifically. Projects with a practice to coordinate work through issues upfront have a higher rate of developers who attempt to integrate their changes ($\mathbf{H}_3$) and have a higher rate of accepted pull requests ($\mathbf{H}_4$).

However, coordination is known to incur some costs and could potentially be annoying to some. Our models provide support for these concerns, suggesting that higher levels of coordination might actually encourage hard forks ($\mathbf{H}_8$). Again, note a similar tradeoff as with modularity, albeit this time more clearly visible in our models: coordination is directly and positively ($\mathbf{H}_8$) associated with likelihood of hard forking, but also indirectly and negatively ($\mathbf{H}_4$), through its effect on pull request acceptance rates (hard forks are associated more with projects that are more selective in accepting external pull requests; $\mathbf{H}_6$). We suspect that developers have to make deliberate tradeoff decisions about how inclusive they want to be in accepting community contributions, potentially at the cost of discouraging contributors and fragmenting their community if their standards are too rigid.

Redundant development. Finally, our models of duplicate pull requests are not sufficiently well fitting to conclude there is strong evidence supporting different interventions; we found some evidence, but weaker compared to the other hypotheses, that claiming an issue upfront is associated with a lower chance of redundant work ($\mathbf{H}_5$). Duplicates are rare in most projects, but may still cause substantial friction, especially for new developers; also, despite many recommendations, claiming issues is not a common practice yet in most projects. Interestingly, anecdotally, we found cases where developers triggered duplicate work by posting an issue *before* addressing the issue themselves without actually claiming the issue, which encouraged others to work on the same issue in parallel. More research is needed to develop and evaluate interventions. Recently suggested awareness tools that might detect duplicate work quickly rather than expecting upfront coordination [49, 67] might be an interesting alternative strategy.

5.2 Implications

Implications for practitioners. Our results encourage practitioners to strive for implementations that are modularly extensible and to adopt guidelines for contributors that suggest coordinating planned changes through an issue tracker. Though some open-source developers might dislike the rigidness and effort of central coordination, our results show that projects that do so receive a higher fraction of pull requests from their active forks, end up integrating more changes, and likely frustrate fewer contributors in the process. Maintainers might want to point newcomers especially to work on problems which can be completed with modular changes. All of this can improve sustainability and the perception of having a strong community for a project. Finally, while hard forks are rare in practice, they can be expensive for a community and have gotten much easier on social coding platforms—maintainers should consider carefully to what degree they can remain open to various external contributions and how modularity can help to integrate contributions more easily or to what degree they are willing to accept some degree of fragmentation.

Implications for researchers and tool builders. While we explored how project characteristics and existing practices influence efficiency outcomes, there are many opportunities to design and study further interventions. For example, improved tooling to navigate and understand changes in forks or to oversee large numbers of pull requests [3–5, 66, 84] can help both maintainers and contributors to explore not-integrated forks and detect work in progress, to detect interesting extensions and avoid redundant development. Explicit GitHub mechanisms rather than conventions to claim issues as work in progress have been suggested [2], as have community tooling for coordination [3], which would be worth evaluating. There may be research opportunities to detect redundant pull requests automatically to reduce the maintainers' effort [49, 67, 83] or even to detect redundant development early before developers finished their work [67]. Research on mentoring [20, 31] might further establish good and efficient practices.

Furthermore, we suspect that many members of an open-source community are not aware of their practices and how they relate to other projects (e.g., some interviewees where surprised that some projects largely coordinate work in the issue tracker whereas others were surprised that not all projects do that). We suspect that making practices transparent, for example, through *repository badges* [76] or *metric dashboards* [16, 19] can help community members to understand their practices and how it relates to other (possibly more efficient) projects.

Finally, we argue that researchers should revisit hard forks and the cost of community fragmentation, given that new ease of forking on social-coding platforms may have changed dynamics from the feared hard forks of the past. Many tools to manage distributed development with forks can also be useful for industrial settings, where forks are also frequently used for collaboration and for variant management [27], and recently several researchers have explored lightweight tooling to support fork-based variant management [7, 32, 71].

6 CONCLUSION

Fork-based development in social-coding context has been widely adopted in open-source communities, as it allows developers to modify their own fork without affecting others and provides a uniform way of contribution their changes back to the original project. We show that there are significant inefficiencies in the collaborative development process of many communities, including lost contributions, rejected pull requests, redundant development, and fragmented communities. Through large-scale statistical modeling of factors operationalized in GitHub traces, we found that many of these inefficiencies associate with common project characteristics and practices, especially modularity and coordination practices.

Acknowledgements. Many thanks to participants of our interviews! Kästner and Zhou have been supported in part by the NSF (awards 1318808, 1552944, and 1717022) and AFRL and DARPA (FA8750-16-2-0042). Vasilescu has been supported in part by the NSF (award 1717415). We thank James D. Herbsleb and Laura Dabbish for their comments and advice on this project.

REFERENCES

[1] 2011. Requirement of "Claiming" tickets in Django project. https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/submitting-patches/#claiming-tickets

[2] 2016. Dear Github Issue 191: Feature: Work In Progress Pull Requests. https://github.com/dear-github/dear-github/issues/191

[3] 2016. WIP app for GitHub. https://github.com/apps/wip

[4] 2017. GitHub Pull Request Triage. http://prs.mozilla.io/

[5] 2017. Lovely Forks Browser Extension: Show notable forks of Github repositories under their names. https://github.com/musically-ut/lovely-forks

[6] 2019. Replication Package. https://doi.org/10.5281/zenodo.3258821

[7] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Ștefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *Comp. Int'l Conf. Software Engineering (ICSE)*. ACM, 532–535.

[8] Amirhosein Emerson Azarbakht. 2017. *Longitudinal Analysis of Collaboration in Forked Open Source Software Development Projects*. Ph.D. Dissertation. Oregon State University.

[9] Carliss Young Baldwin and Kim B Clark. 2000. *Design Rules: The Power of Modularity*. Vol. 1. MIT press.

[10] Fabian Beck and Stephan Diehl. 2011. On the Congruence of Modularity and Code Coupling. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 354–364.

[11] Andrew Begel and Thomas Zimmermann. 2014. Analyze this! 145 questions for data scientists in software engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 12–23.

[12] Mary J Benner and Michael L Tushman. 2003. Exploitation, exploration, and process management: The productivity dilemma revisited. *Academy of management review* 28, 2 (2003), 238–256.

[13] Marco Biazzini and Benoit Baudry. 2014. May the Fork be with You: Novel Metrics to Analyze Collaboration on GitHub. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 37–43.

[14] James M Bieman and Linda M Ott. 1994. Measuring functional cohesion. *IEEE Trans. Softw. Eng. (TSE)* 20, 8 (1994), 644–657.

[15] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM, 109–120.

[16] E. Bouwers, A. van Deursen, and J. Visser. 2013. Evaluating usefulness of software metrics: An industrial experience report. In *Proc. Int'l Conf. Software Engineering (ICSE)*. 921–930.

[17] Jordi Brandts and David J. Cooper. 2018. *Truth Be Told An Experimental Study of Communication and Centralization*. Working Papers 1046. Barcelona Graduate School of Economics.

[18] Yuangfang Cai and Sunny Huynh. 2007. An evolution model for software modularity assessment. In *Proc. ICSE Workshop on Software Quality (WoSQ)*. IEEE, 3–3.

[19] G. Ann Campbell and Patroklos P. Papapetrou. 2013. *SonarQube in Action* (1st ed.). Manning Publications Co., Greenwich, CT, USA.

[20] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. Who is Going to Mentor Newcomers in Open Source Projects?. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM, Article 44, 11 pages.

[21] Bee Bee Chua. 2017. A Survey Paper on Open Source Forking Motivation Reasons and Challenges. In *21st Pacific Asia Conference on Information Systems (PACIS)*. 75.

[22] Melvin E Conway. 1968. How do committees invent. *Datamation* 14, 4 (1968), 28–31.

[23] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. ACM, 1277–1286.

[24] Laura Dabbish, Colleen Stuart, Jason Tsay, and James Herbsleb. 2013. Leveraging transparency. *IEEE Software* 30, 1 (2013), 37–43.

[25] Carlo Daffara. 2012. Estimating the economic contribution of open source software to the European economy. In *The First Openforum Academy Conference Proceedings*. 11–14.

[26] Premkumar Devanbu, Thomas Zimmermann, and Christian Bird. 2016. Belief & evidence in empirical software engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 108–119.

[27] Yael Dubinsky, Julia Rubin, Theodore Berger, Slawomir Duszynski, Matthias Becker, and Krzysztof Czarnecki. 2013. An exploratory study of cloning in industrial software product lines. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[28] Nadia Eghbal. 2016. *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure*. Ford Foundation. https://www.fordfoundation.org/media/2976/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure.pdf

[29] Thomas J Emerson. 1984. A discriminant metric for module cohesion. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, 294–303.

[30] Neil A Ernst, Steve Easterbrook, and John Mylopoulos. 2010. Code forking in open-source software: a requirements perspective. *arXiv preprint arXiv:1004.2889* (2010).

[31] Fabian Fagerholm, Alejandro S Guinea, Jürgen Münch, and Jay Borenstein. 2014. The role of mentoring and project characteristics for onboarding in open source software projects. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM, 55.

[32] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing clone-and-own with systematic reuse for developing software variants. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 391–400.

[33] Brian Fitzgerald. 2005. Has open source software a future. *Perspectives on free and open source software* 1 (2005), 93–106.

[34] Karl Fogel. 2005. *Producing open source software: How to run a successful free software project*. O'Reilly Media, Inc.

[35] Kam Hay Fung, Aybüke Aurum, and David Tang. 2012. Social Forking in Open Source Software: An Empirical Study. In *Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE) Forum*. Citeseer, 50–57.

[36] Jonas Gamalielsson and Björn Lundell. 2014. Sustainability of Open Source Software Communities beyond a Fork: How and Why has the LibreOffice Project Evolved? *Journal of Systems and Software* 89 (2014), 128–145.

[37] Georgios Gousios. 2013. The GHTorent dataset and tool suite. In *Proc. Working Conf. Mining Software Repositories (MSR)*. IEEE Press, 233–236.

[38] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 345–355.

[39] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *Proc. Int'l Conf. Software Engineering (ICSE)*, Vol. 1. 358–368.

[40] Shane Greenstein and Frank Nagle. 2014. Digital dark matter and the economic contribution of Apache. *Research Policy* 43, 4 (2014), 623–631.

[41] Carl Gutwin and Saul Greenberg. 2004. The Importance of Awareness for Team Cognition in Distributed Collaboration. *E. Salas & S. M. Fiore (Eds.), Team cognition: Understanding the factors that drive process and performance* (2004), 177–201.

[42] Carl Gutwin, Reagan Penner, and Kevin Schneider. 2004. Group awareness in distributed software development. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. ACM, 72–81.

[43] James D Herbsleb and Rebecca E Grinter. 1999. Splitting the organization and integrating the code: Conway's law revisited. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 85–95.

[44] James D. Herbsleb and Audris Mockus. 2003. An empirical study of speed and communication in globally distributed software development. *IEEE Trans. Softw. Eng. (TSE)* 29, 6 (2003), 481–494.

[45] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. 2017. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering* 22, 1 (2017), 547–578.

[46] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M German. 2015. Open source-style collaborative development practices in commercial projects using GitHub. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, 574–585.

[47] Bruce Kogut and Anca Metiu. 2001. Open-source software development and distributed innovation. *Oxford review of economic policy* 17, 2 (2001), 248–264.

[48] Andrew M St Laurent. 2004. *Understanding Open Source and Free Software Licensing: Guide to Navigating Licensing Issues in Existing & New Software*. O'Reilly Media, Inc.

[49] Zhixing Li, Gang Yin, Yue Yu, Tao Wang, and Huaimin Wang. 2017. Detecting Duplicate Pull-requests in GitHub. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*. ACM, 20.

[50] Alec Liu. 2014. Who's Building Bitcoin? An Inside Look at Bitcoin's Open Source Development. *Motherboard* (2014).

[51] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. 2006. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science* 52, 7 (2006), 1015–1030.

[52] James G March. 1991. Exploration and exploitation in organizational learning. *Organization science* 2, 1 (1991), 71–87.

[53] Ines Mergel. 2015. Open collaboration in the public sector: The case of social coding on GitHub. *Government Information Quarterly* 32, 4 (2015), 464–472.

[54] Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Vol. 2. Prentice hall New York.

[55] Vishal Midha and Prashant Palvia. 2012. Factors Affecting the Success of Open Source Software. *J. Syst. Softw.* 85, 4 (April 2012), 895–905.

[56] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. 2002. Evolution patterns of open-source software systems and communities. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*. ACM, 76–85.

[57] Linus Nyman. 2014. Hackers on forking. In *Proc. Int'l Symposium on Open Collaboration (OpenSym)*. ACM, 6.

[58] Linus Nyman and Tommi Mikkonen. 2011. To fork or not to fork: Fork motivations in SourceForge projects. In *IFIP International Conference on Open Source Systems*.

Springer, 259–268.

[59] Linus Nyman, Tommi Mikkonen, Juho Lindman, and Martin Fougère. 2012. Perspectives on Code Forking and Sustainability in Open Source Software. *Open Source Systems: Long-Term Sustainability* (2012), 274–279.

[60] Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. 2011. Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, Vol. 6813. Springer, 155–178.

[61] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[62] Phanish Puranam, Oliver Alexy, and Markus Reitzig. 2014. What's "new" about new forms of organizing? *Academy of Management Review* 39, 2 (2014), 162–180.

[63] Ayushi Rastogi and Nachiappan Nagappan. 2016. Forking and the Sustainability of the Developer Community Participation–An Empirical Investigation on Outcomes and Reasons. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 102–111.

[64] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, 367–377.

[65] Eric S Raymond. 2001. *The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary.* O'Reilly Media, Inc.

[66] Luyao Ren, Shurui Zhou, and Christian Kästner. 2018. Poster: Forks Insight: Providing an Overview of GitHub Forks. In *Comp. Int'l Conf. Software Engineering (ICSE)*. ACM, 179–180. Poster.

[67] Luyao Ren, Shurui Zhou, Christian Kästner, and Andrzej Wąsowski. 2019. Identifying Redundancies in Fork-based Development. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 230–241.

[68] Gregorio Robles and Jesús M. González-Barahona. 2012. A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes. In *Open Source Systems: Long-Term Sustainability International Conference, OSS.* 1–14.

[69] Maha Shaikh and Ola Henfridsson. 2017. Governing open source software through coordination processes. *Information and Organization* 27, 2 (2017), 116–135.

[70] Jesse Shore, Ethan Bernstein, and David Lazer. 2015. Facts and figuring: An experimental investigation of network structure and performance in information and solution spaces. *Organization Science* 26, 5 (2015), 1432–1446.

[71] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, operations, and feasibility of a projection-based variation control system. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 323–333.

[72] Igor Steinmacher, Gustavo Pinto, Igor Scaliante Wiese, and Marco Aurélio Gerosa. 2018. Almost there: A study on quasi-contributors in open-source software projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 256–266.

[73] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*. 151–160.

[74] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, 107–119.

[75] Linus Torvalds. 1999. The Linux Edge. *Commun. ACM* 42, 4 (1999), 38–38.

[76] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding Sparkle to Social Coding: An Empirical Study of Repository Badges in the Npm Ecosystem. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, 511–522.

[77] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 356–366.

[78] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 644–655.

[79] Robert Viseur. 2012. Forks impacts and motivations in free and open source projects. *International Journal of Advanced Computer Science and Applications* 3, 2 (2012), 117–122.

[80] Jian Wang. 2016. Knowledge creation in collaboration networks: Effects of tie configuration. *Research Policy* 45, 1 (2016), 68–80.

[81] Steve Weber. 2004. *The success of open source.* Harvard University Press.

[82] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. 2015. Wait for it: Determinants of pull request evaluation latency on GitHub. In *Proc. Working Conf. Mining Software Repositories (MSR)*. IEEE, 367–371.

[83] Yue Yu, Li Zhixing, Yin Gang, Tao Wang, and Wang Huaimin. 2018. A Dataset of Duplicate Pull-requests in GitHub. In *Proc. Working Conf. Mining Software Repositories (MSR)*. 12.

[84] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM Press, 105–116.

[85] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. 2003. How history justifies system architecture (or not). In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*. IEEE, 73–83.