Improving Collaboration Efficiency in Fork-based Development

Shurui Zhou

May 2020
CMU-ISR-20-103

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christian Kästner (Advisor)
James D. Herbsleb
Laura A. Dabbish
Andrzej Wąsowski (IT University of Copenhagen)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This research was sponsored by the National Science Foundation (awards 1318808, 1552944, and 1717022) and AFRL and DARPA (FA8750-16-2-0042). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Center for the Future of Work or the National Science Foundation

Keywords: Collaborative Software Development, Distributed Collaboration, Fork-Based Development, Social Coding, GitHub, Open-Source

Abstract

Fork-based development is a lightweight mechanism that allows developers to collaborate with or without explicit coordination. Recent advances in distributed version control systems (e.g., '*git*') and social coding platforms (e.g., GitHub) have made fork-based development relatively easy and popular by providing support for tracking changes across multiple forks with a common vocabulary and mechanism for integrating changes back. However, fork-based development has well-known downsides. When developers each create their own fork and develop independently, their contributions are usually not easily visible to others, unless they make an active attempt to merging their changes back into the original project. When the number of forks grows, it becomes very difficult to keep track of decentralized development activity in many forks. The key problem is that it is difficult to maintain an overview of what happens in individual forks and thus of the project's scope and direction. Furthermore, the problem of lacking an overview of forks can lead to several additional problems and inefficient practices: lost contributions, redundant development, fragmented communities, and so on.

In this dissertation, I mixed a wide range of research methods to understand the problem space and the solution space. Specifically, I first design measures to quantify how serious are these inefficiencies, then I developed two complementary strategies to alleviate the problem: First, during the process of sampling 1311 GitHub projects and quantifying the inefficiencies, also by opportunistically reaching out to developers who have used forks, I recognized that there are differences among projects. Therefore, I identified existing best practices and suggesting evidence-based interventions for projects that are inefficient. Moreover, I observed that the notion of forking has changed since the invention of fork-based development, so I conducted mixed-method experiment to understand the perception of forking by interviewing developers and identified future research directions. Second, as we found that the *lack of an overview* problem that we observed in fork-based development environment is essentially the same as the *lack of awareness* problem that have been studied previously in other distributed software development scenarios but with new challenges, I designed awareness tool to improve the awareness in the fork-based development environment and help developers to detect redundant development to reduce developers' unnecessary effort. To evaluate the effectiveness and usefulness of these awareness tools, I conducted both quantitative and qualitative studies.

My dissertation work focuses on improving collaboration efficiency for distributed software teams, but the research method has a lot wider applicability. For example, in the future, I will study other forms of collaboration, such as the collaboration of interdisciplinary software teams.

Acknowledgments

I would not have finished this thesis without the help and support from my professors, my family, and my friends, and my colleagues.

First and foremost, I would like to thank my advisor, Prof. Christian Kästner, who is always supportive and patient, provides me guidance, and challenges me. I could have never been here without his countless and significant support for the last 6 years. Through him, I learned the spirit of academics. I will never forget when I was struggling with research, mentoring experiences, and my future plans, he said "I am here to help". I wish I could be a great advisor as Christian in the future.

I would like to express my thanks to my 'informal' advisor and collaborator Prof. Bogdan Vasilescu, who is always positive, supportive, and optimistic. From him, I learned the power of passion and perseverance.

I would like to say 'thank you' to Prof. Andrzej Wąsowski, who is a terrific collaborator, a mentor, and has been helping me since 2015, and always provides me valuable and helpful feedback.

I would like to thank my thesis committee members Prof. James D. Herbsleb and Prof. Laura A. Dabbish. I have learned a great deal from this joyful experience.

I am so grateful to Prof. Yingfei Xiong at Peking University. Without Yingfei's support, I would not have the chance to join Carnegie Mellon. Without his help and willing to sacrifice rest time over 2 years to attend my weekly meetings at night, I could not publish my first ICSE paper and get the motivation of pursuing my academic dream.

I would like to express my deepest gratitude to Prof. Yuan Rao at Xi'an Jiaotong University who helped me, supported me, guided me for the last 12 years, in Xi'an and in Pittsburgh, as an advisor, a friend and family.

I would like to thank my professors in ISR – Prof. Claire Le Goues, Prof. Eunsuk Kang, Prof. Fei Fang, Prof. David Garlan, Prof. Michael Hilton, Prof. Heather Miller, who gave me millions of valuable advice and help during my Ph.D.

I would like to give thanks to my wonderful and perfect parents Xiling Wang and Lingguo Zhou, who give me endless and selfless love and always cheer me on, just as they have every step of the way. They gave me the ability to maintain hope for a brighter morning, even during our darkest night. It is time to repay my parents.

I am so lucky to have my husband Xilin Liu, who is my knight in shining armor and a gift. He always supports me and has been amazing during my Ph.D. Xilin, we are best friends, best buddies, and you can always count on me.

I would like to thank my best friend Shengchen Du, who has been the definition of what a friend is. For the past 12 years, you always support me and trust me no matter whether we are in the same city or miles apart. I promise I will be supportive to you as always and treasure our friendship like my favorite lyrics in Friends – "I'll be there for you, like I've been there before; I'll be there for you, because you're there for me too".

Special thanks to my friend and colleague, Chu-Pan Wong, who is always positive and reliable. Thank you for being on this journey with me. I wish you all the best in your future.

I would like to thank my brilliant girls and roommates, Zhuyun Dai and Fuchen Liu. Without their support, my life in Pittsburgh would be boring and lonely. Thank you for sticking by me.

I would like to thank my kindred spirit, Ang Liu, who is always by my side and would explore this brave new world with me.

I would like to also thank my officemates and colleagues, Jaspreet Bhatia, Hemank Lamba, Ştefan Stănciulescu, Gabriel Ferreira, Jens Meinicke, Pooyan Jamshidi, Miguel Velez, Olaf Leßenich, Connie Herold, Jamie Lou Hagerty, and Jennifer Cooper, and all other people who shared their expertise and great ideas and always provided me prompt support so that I could enjoy such an open and collaborative environment.

Thank you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this dissertation, I study how to improve collaboration efficiency for distributed software teams using fork-based development mechanism. Fork-based development is a innovation that provides developers the flexibility to implement ideas without affecting each other and has completely changed the way of collaboratively building software systems, however it has downsides. For example, when the number of forks increases, developers find it is difficult to maintain an overview of the activities of the team, which further leads to collaboration inefficiencies like lost contribution, redundant development, and fragmented community (shown as the **problem** space in Figure 1.1). Facing these problems, I design complementary solutions to address corresponding problems (shown as the **solution** space in Figure 1.1): First, I study natural interventions to identify best practices; Second, I design new interventions to improve the fork-based development mechanism. Last but not least, I mix a wide range of research method to evaluate the solutions from different perspectives.

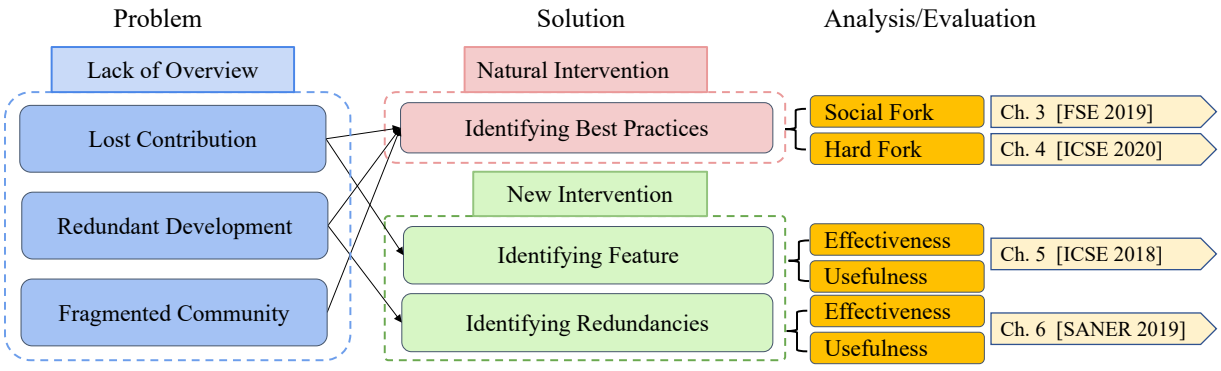


Figure 1.1: Outline of the thesis. (Arrows present the mapping between solution and its targeting problems)

Collaboration is essential for software development at scale, in both industrial and open-source projects. As the software teams become increasingly distributed, many of the mechanisms that support coordination in a co-located setting are absent or disrupted in a distributed project [103, 106, 139]. Geographic distance profoundly affects the ability to collaborate [165],

and leads to various disruptions to different degrees, such as much less communication, lack of awareness, and incompatibilities [106].

Fork-based (known as branch-based or pull-based) development is an emerging paradigm and a lightweight mechanism that supports distributed software development. Developers could start with an independent development from an existing codebase by simply copying code files and creating a fork or a branch [36], while having the freedom to make any modifications [37, 71, 81, 230]. Recent advances in distributed version control systems [2, 223] (e.g., '*git*') and social-coding platforms (e.g., GITHUB, Bitbucket, and GITLAB) have made fork-based development relatively easy and popular [97, 179] by providing support for tracking changes across multiple forks, and using a common vocabulary and mechanism for integrating changes back [61]. More and more projects, both closed source and open source, are being migrated to these code hosting sites [24]. As of January 2020, GitHub reports having over 40 million users and more than 100 million repositories (including at least 28 million public repositories), making it the largest host of source code in the world [14].

Forking has become very common: As we measured from the GHTorrent [96] data, 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. The large population of forks in the distributed software development comes at costs to open-source (such as lacking of an overview, lost contribution, redundant development, and fragmented communities), which may even threaten the sustainability of the open-source communities, as we will explain later. Moreover, inadequate models of collaboration can stifle innovation, hurt common infrastructure, and lead to inefficient development process, for example, when team members lack of awareness of what others are doing [70, 103], when code structure does not align with team structure [107], or when the structure of governance of a community is inefficient [64]. Improving this situation is the core goal of this thesis.

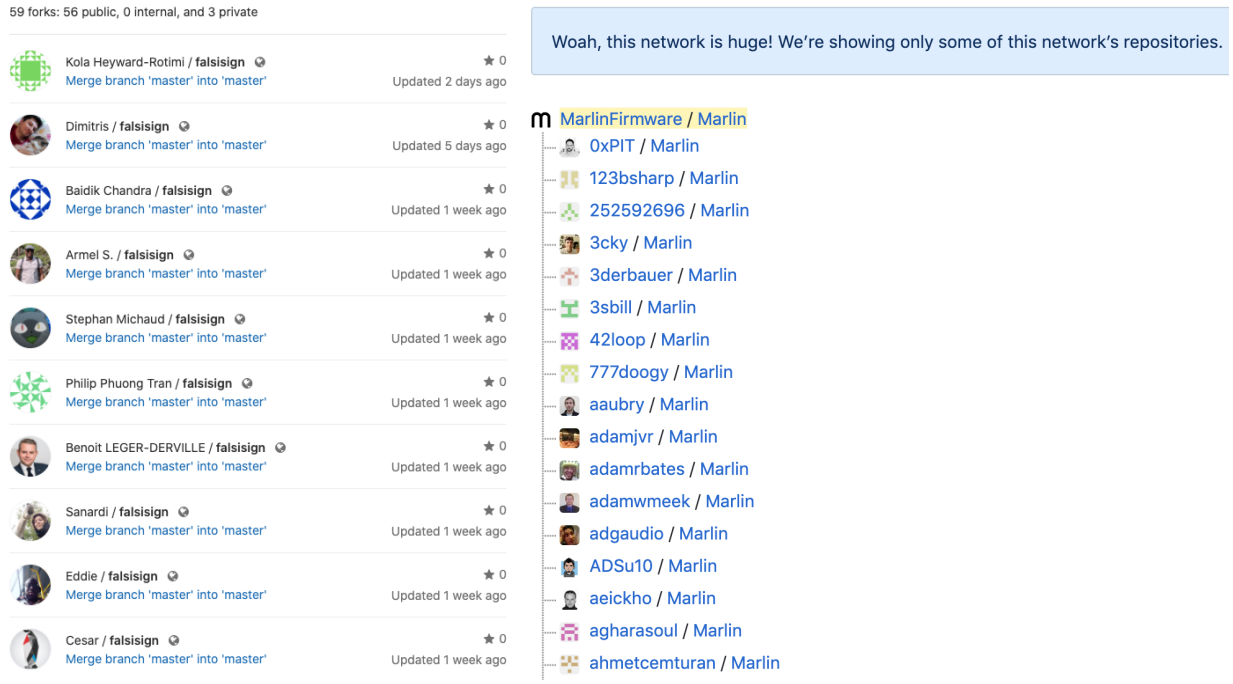
Before the rise of social coding, forking traditionally referred to the intention of splitting an independent development line, competing with the original repository, often with a new name. We use the term **(social) fork** in the sense of creating a public copy of a git repository and refer to the traditional definition of splitting of a new independent project as a **hard fork**. We explain the history of forking in Section 2.1.

1.1 Inefficiencies in Social Forking

Modern tools and platforms (*e.g.*,GITHUB, Bitbucket, and GITLAB) have made forking easier (1) to track and integrate changes across multiple forks without central management and (2) to publish changes, including incomplete and experimental ones. Forking has become very common and popular [97, 179] as we described previously. Social forking has been broadly studied from different perspectives [61, 62, 97, 99, 100, 147, 226].

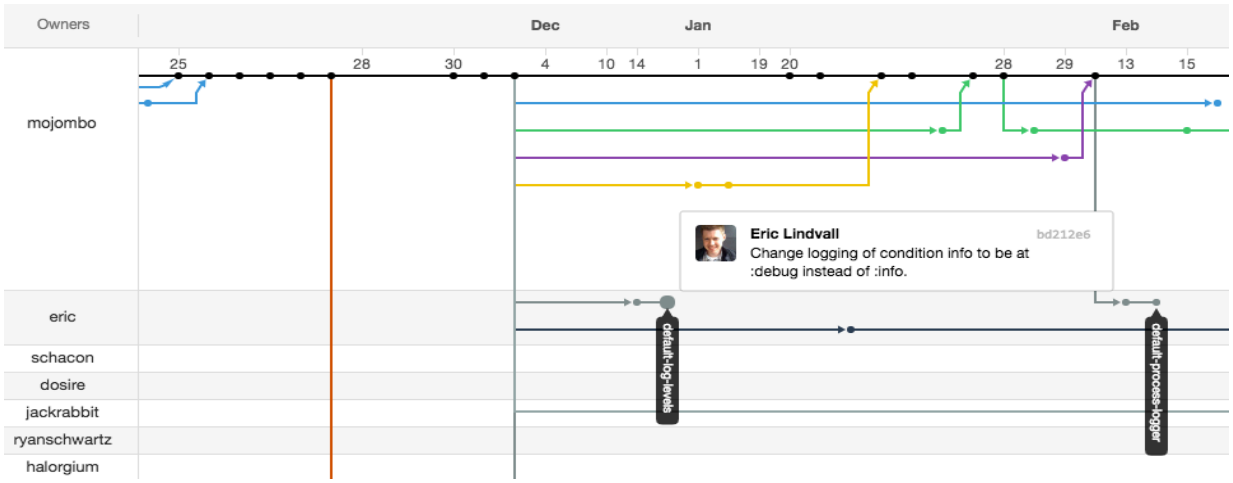
While easy to use and popular in practice, fork-based development has well-known downsides. When developers each create their own fork and develop independently, their contributions are usually not easily visible to others, unless they make an active attempt to merge their code changes back into the original project. When the number of forks grows, it becomes difficult to keep track of decentralized development activity in many forks. The key problem is that it is *dif-*

ficult to maintain an overview of what happens in individual forks and thus of the project's scope and direction. Also, for fork-based development in industrial contexts, both Berger et al. and Duc et al. found that it is hard for individual teams to know who is doing what, which features exist elsewhere, and what code changes are made in other forks [28, 72].



(a) GITLAB's fork list view.

(b) GITHUB's fork list view.



(c) GITHUB's network graph shows commits across known forks, but is difficult to use to gain an overview of activities in projects with many forks [249].

Figure 1.2: Existing solutions for the problem of lack of overview in fork-based development.

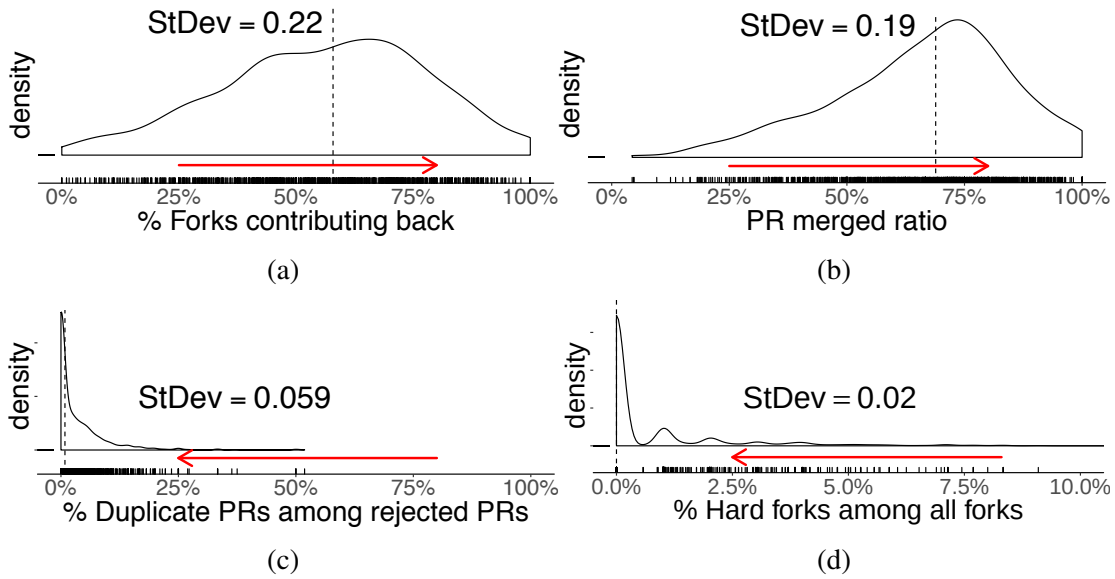


Figure 1.3: Among 1311 GITHUB projects, both efficient and inefficient forking practices are common, motivating us to understand what influences inefficiencies. Plots are density plots indicating which outcomes are common in many projects; the arrow indicates more efficient outcomes; the dash line indicates the median [250].

Open-source developers (including the ones that we interviewed for this project) indicated that they are interested in what happens in other forks [231], but cannot effectively explore them with current technologies [7]. Code hosting platforms invented different solutions to resolve this problem. For example, GITHUB and GITLAB list all the forks of a project in one page (see Figure 1.2a and Figure 1.2b). In addition, GITHUB's network graph shown in Figure 1.2c visualizes the history of commits over time across all branches and forks of a project [61]. Although the fork list view helps people to know the existing forks, it is hard to figure out whether these forks are still active and what are the code changes. While the network view is a good starting point to understand how the project evolves, it is tedious and time consuming to use if a project has many forks. In order to see older history, users click and drag within the graph, and if users want to see the commit information, they hover the mouse over each commit dot and read the commit message. Also, they complain that they *"have to scroll back a lot to find the fork point and then go to the end again for seeing what changed since then in the parent and in the fork"* [7]. If developers want to investigate the code changes of certain forks, they have to manually open and check each fork. The GITHUB network view does not even load when there are over 1000 forks, no matter whether they are active or inactive. As we measured from the GHTorrent [96] data, over 2,236 GITHUB projects have more than 1000 forks as of June 2019.

Furthermore, the **lack of overview** of forks can lead to several additional problems and **inefficient practices**:

- ***Lost contributions:*** Developers may explore interesting ideas, fix bugs, or add useful features in forks, but unless they contribute those changes back to the original project, those contributions are easily lost to the larger community, although these changes are technically public [249]. Fung et al. [91] report that only 14 percent of all active forks of

nine popular JavaScript projects on GITHUB integrated back any changes; extrapolating to open-source in general, this can amount to significant inefficiencies regarding development talent and lost effort. Developers are often interested in activities by other developers, but simply are not able to follow details in that many forks proactively—for example, one of the developers in our user study discovered a feature in a fork and said *"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 many forks [12, 184, 249]. In our study, we regard a community in which more developers *attempt to contribute* their changes to the upstream as more efficient. In our sample of 1311 GITHUB projects, we identified the fraction of forks that attempt to contribute any changes back among all active forks. And results shown that this problem is pervasive: a median of 50% active forks never contribute back to the upstream (see Figure 1.3a). Lost contributions even happen in coordinated software product line when people often struggle to identify which of multiple existing forks/branches to select as a starting point [71].

- ***Rejected pull requests.*** Not all attempted contributions are accepted by project maintainers. When developers submit a pull request that gets rejected, they can perceive this as a waste of their effort and get discouraged from contributing further [212]. One important factor that affects the decision of merging a pull request or not is project fit, which means whether the proposed pull request is in line with the goals and target of the project, and also the technical fit – does the code fit the technical design of the project [18, 100, 212]. From the community's perspective, a project in which most pull requests are accepted can be considered as most effective with regard to contributor efforts. Observing the rate of rejected pull requests among all closed pull requests in our 1311 GITHUB projects (see Figure 1.3b), we see that in most projects a majority of pull requests 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 pull requests whereas others accept only a small percentage, and how project maintainers can strive for more efficiency.
- ***Redundant development:*** Unaware of activities in other forks, developers may re-implement functionality already developed elsewhere. A developer we interviewed in our study (Chapter 5) [249] also confirmed the problem as follows: *"I think there are a lot of people who have done work twice, and coded in completely different coding style."* Gousios et al. [97] summarized nine reasons for rejected pull requests in 290 projects on GITHUB, in which 23% were rejected due to redundant development (either parallel development or superseded other pull requests). Redundant development further leads to merging conflicts, which would demotivate or prevent developers from continuously contributing to the repository [97, 212] and significantly increases the maintenance effort for maintainers [71, 214]. In analyzing the fraction of pull requests rejected due to redundancies, we found that redundant development is a small but pervasive problem: about 1–5 % of all pull requests and 5–50 % of rejected pull requests (see Figures 1.3c).
- ***Fragmented Communities:*** Diffusion of efforts can be observed on GITHUB in the many secondary forks (*i.e.*, forks of forks) that contribute to other forks, but not to the original repository [91, 211]. This fragmentation can seriously threaten the sustainability of open source projects when scarce resources are additionally scattered across multiple projects. In fragmented communities, we see multiple related repositories receive contributions,

but those contributions are rarely shared. For example, *Ultimaker* was originally a fork of the *Marlin* project aimed at certain hardware, but has evolved into an independently managed hard fork with over 190 own forks and no interaction with *Marlin* anymore; inefficiencies can be observed, for example, in a pull request for Marlin for an issue that was independently fixed with a different pull request in *Ultimaker* two years earlier.[1] There are different reasons for community fragmentation. For example, one of the developers who have second level forks explained that he implemented a feature and submitted a pull request. After it was rejected, he started to focus on his fork, and then more and more developers started to fork his fork. Hard forks are rare, but potentially expensive for a community. In analyzing the percentage of hard forks among all the sampled forks of each project, the numbers show that a median of 5% sampled projects have hard forks (see Figure 1.3d). Although this only happens to some projects, but the problem is severe.

Note that **not all the practices that we described above (including unmerged code changes in forks, rejected pull requests, similar implementations from different developers, and fragmented communities) are inefficient and should be eliminated**. For example, not all of the unmerged code changes in forks are reusable, and they could be changes done for some other reasons, like experimenting, getting familiar with the project, or for customization. It might be more efficient not to send the owner a pull request that would take up the owner's time and effort to review a useless change. Moreover, these practices could even be beneficial. For example, duplication could stimulate better solutions if two developers communicate and collaborate earlier. However, currently there is no tool to notify developers who are working on similar features or bug-fixes. Therefore, instead of collaborating upfront, developers compete at the end. Similarly, community fragmentation could be beneficial for exploring new and larger ideas or testing whether there is sufficient support for features and ports for niche requirements or new target audiences (see Section 4.3.2). However, current technology does not support coordination across multiple hard forks well.

We have evidence showing that a large portion of communities treat these as inefficiencies, and would like to address these to some extent (see Chapter 3). And studies have found that there are many chances that developers could be notified about these cases earlier and collaborate more efficiently (Chapter 5 and 6). Therefore, as the first step, in this thesis, we treat *lost contribution, rejected pull requests, redundant development, and fragmented communities* as indicators of collaboration inefficiencies and design complementary solutions to address them. In the future, we could design approaches to detect the intention behind each developer's activity and come up with a more targeted method to help community members to achieve higher collaboration efficiency, but this is out of the scope of this thesis.

[1] `https://github.com/MarlinFirmware/Marlin/pull/10119`
`https://github.com/Ultimaker/Ultimaker2Marlin/pull/118`

1.2 Possible Solutions

We would like to alleviate these inefficiencies. We developed two complementary strategies: Identifying natural interventions and designing new interventions.[2] First, by quantifying the inefficiencies in a large number of GITHUB projects and by discussing with multiple developers regarding their experiences of using forks, we recognized that there are differences among projects in terms of the degree of inefficient practices. These strong differences raise the question of why these projects are so different and whether we could find natural interventions that already exist and are correlated with higher collaboration efficiency, and then recommend these best practices to other communities with lower collaboration efficiency. Second, since there is a lot of information that is publicly available but not easily accessible, we saw opportunities of building awareness tools to help people to gain a better understanding of the activities of others and a context for their own activity in collaborative software development.

Identifying natural interventions. During discussions with developers from different open-source communities, we observed that some communities have fewer problems with inefficiencies than others. In quantifying the inefficiencies of our sampled projects set, we found that projects are indeed very different regarding the degree of collaboration inefficiencies. Specifically, the degree of fork owners attempting to contribute their changes back to the upstream varies, ranging from projects in which almost no fork attempts to contribute back, to projects where almost all forks are used for attempted contributions (see Figure 1.3a). Figure 1.3b shows that in most projects a majority of pull requests are accepted, but again with large differences. Similarly, by plotting the fraction of pull requests rejected due to redundancies in Figures 1.3c, we observe the differences that some projects have more redundant development cases than others. And Figure 1.3d shows the percentage of hard forks among all the sampled forks of each project, we again observe that their frequency varies significantly across projects.

These strong differences in observed inefficiencies raise the question of why these projects are so different, and bring us the opportunity of improving collaboration efficiencies for open-source communities by identifying actually occurring interventions from some projects that are more efficient.

Designing new interventions to improve awareness. As the fork-list view (GITHUB and GITLAB) and the network view (GITHUB) shown in Figure 5.1 are not good enough to provide developers an overview of the activities in the community, we would like to design new interventions to improve current situation. Through literature analysis, we found that the *lack of an overview* problem that we observed in fork-based development environment is essentially the same as the *lack of awareness* problem that have been studied previously in other distributed software development scenarios [48, 63, 70, 103, 200, 224], but with new challenges.

As there is a lot of information that is publicly available but not easily accessible in the fork-based development environment, we saw opportunities for designing new intervention – building awareness tools – to help team members to gain an understanding of the activities of others,

[2]The term intervention is used in social studies and social policy to refer to the decision making problems of intervening effectively in a situation in order to secure desired outcomes [3].

which also provides a context for each developer's own activity [70], and ultimately mitigate the collaboration inefficiency. Specifically, we designed an approach INFOX [249] to summarize un-merged code changes in forks in order to generate a better overview of the community. We also designed an approach INTRUDE [185] to identify potentially redundant code changes to save both project maintainers' and developers' effort.

Summary. We observe inefficiencies in fork-based development and different communities has different inefficient practices, so we would like to understand *how efficiently developers use forks in different communities, and to what degree project characteristics and practices of open-source communities associate with inefficiencies*. Then we propose two complementary strategies to mitigate those issues: First, we would like to identify existing best practices and suggest evidence-based interventions to projects that are inefficient; second, we would like to build tools that could improve the awareness of a community, and help developers to detect redundant development to unnecessary effort. To evaluate the effectiveness and usefulness of these approaches, we conducted both quantitative and qualitative studies. The research setting for this study is the "social coding" platform GITHUB, which is a very popular contemporary example of a fork-based development environment.

1.3 Thesis

My dissertation work is about alleviating the inefficiencies in fork-based development, by identifying interventions from existing best practices and building awareness tools that could improve the awareness of a community using fork-based development, and reduce developers' unnecessary effort. The following is my **thesis statement**:

I study how communities using forks, design measures to quantify inefficiencies in fork-based development. In order to mitigate the inefficiencies, I propose two strategies: first, I conduct a cross-sectional, correlational study to identify existing best practices and generate evidence-based recommendations that could improve collaboration efficiency; second, I design awareness tools to generate a better overview of code changes in an open-source community, and detect redundant development to reduce waste of maintenance and development effort.

The **research questions** we asked in this thesis are:

- RQ1: What characteristics and practices of a project associate with efficient forking practices?
- RQ2: How have perceptions and practices around hard forks changed?
- RQ3: Can awareness tools help fork-based development to mitigate collaboration inefficiencies?

1.4 Summary of Contribution

The **contributions** of this thesis include the following:

- Measures of inefficiencies in open source communities, and observations of strong differences among projects in terms of lost contributions, rejected pull requests, redundant

development, and fragmented communities. Result shows that projects are different in terms of the degree of collaboration efficiencies (Chapter 2).

- A cross-sectional, correlational study that test hypotheses whether certain context factors of a project are correlated with inefficient practices by fitting statistical models with across 1311 GITHUB projects. The findings show that management strategy of the community and project modularity is correlated with higher efficiency but with trade-offs. Based on the findings, we derived evidence-based guidance to practitioners, and future research directions and tooling ideas (Chapter 3).
- A mix-methods empirical design, combining repository mining with developer interviews to investigate the evolution patterns of hard forks, and study the perceptions of hard forks comparing to 20 years ago. Our finding show that hard forks are a significant concern, even though their relative numbers are low. In addition, we find that the 'stigma' often reported around hard forks is largely gone, indeed forks including hard forks are generally seen as a positive. Furthermore, with social forking encouraging forks as contribution mechanism, we find that many hard forks are not deliberately planned but evolve slowly from social forks (Chapter 4).
- INFOX, an approach and the corresponding tool, which automatically identifies and summarizes features in forks of a project, using source code analysis, community detection, and information retrieval techniques. And we provide evidence that INFOX improves accuracy over existing techniques and provides meaningful insights to maintainers of forks. Furthermore, we developed a web service *forks-insight.com* to improve our research impact in practice (Chapter 5).
- INTRUDE, an approach that automatically identifies duplicate code changes using natural language processing and machine learning. We develop clues for indicating redundant development, beyond just title and description. And we designed quantitative study to prove that the approach outperforms the state-of-the-art (Chapter 6).

1.5 Outline

In this thesis, we first study the problem space by quantifying the efficiencies in fork-based development: Lost contribution, rejected pull requests, redundant development, and fragmented community (already described in Chapter 1). Chapter 2 grounds our work by discussing the history of forking, and the importance of awareness of a distributed collaborative environment.

Next, we propose two complementary strategies to mitigate these problems (Chapter 3-6): First, we identified existing best practices and suggesting evidence-based interventions to projects that are inefficient (marked as the *Natural Intervention* in Figure 1.1); second, we built tools that could improve the awareness of a community using fork-based development and help developers to detect redundant development to reduce developers' unnecessary effort (marked as *New Intervention* in Figure 1.1). To evaluate the effectiveness and usefulness, we conducted both quantitative and qualitative studies. Chapter 7 discusses potential future research directions.

Chapter 2

Distributed Collaboration of Software Development

In this thesis, we discuss different approaches to improving collaboration efficiency for distributed software teams using fork-based development mechanism. In this chapter, we first give a brief introduction of the history of forking and then compare the old notion of forking with the recent social forking phenomenon, which also lays the foundation to one of the projects (Chapter 4) – identifying different types of forks to understand the community fragmentation problem. Furthermore, we discuss the importance of awareness in a collaborative environment, from which we got inspirations and then design new interventions to improve fork-based mechanisms.

2.1 History of Forking

Traditionally, the processes of collaboration in distributed software development is through patch submission and acceptance [34, 35, 156, 235]. With the advances in distributed version control systems (e.g., *git*) and social coding platforms (e.g., GITHUB, GITLAB, and Bitbucket), fork-based development became relatively easy and popular for the last 12 years, both in open-source and in industry [97, 179].

Traditionally, forking was the practice of copying a project and splitting off new *independent* development; in the past, forking was rare and was often intended to compete with or supersede the original project [85, 131, 161, 181]. For example, when *OpenOffice* was acquired by Oracle in 2010 but did not fit anywhere in Oracle's grand plans, developers in the community decided to fork *OpenOffice* and created *LibreOffice* in the same year. Similarly, in 2011, *Hudson* was forked as *Jenkins* because of the governance disagreements with Oracle [9]. In 2014, "after a public spat with the "steward" of the framework" [199, 240], a number of *Node.js* developers started their own fork of the framework called *io.js*, but after a year, the Linux Foundation announced that *Node.js* and *io.js* officially merge codebases back.

Nowadays, forks are typically understood to be 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 distributed version control systems, forking of repositories has become very popular [97, 179].

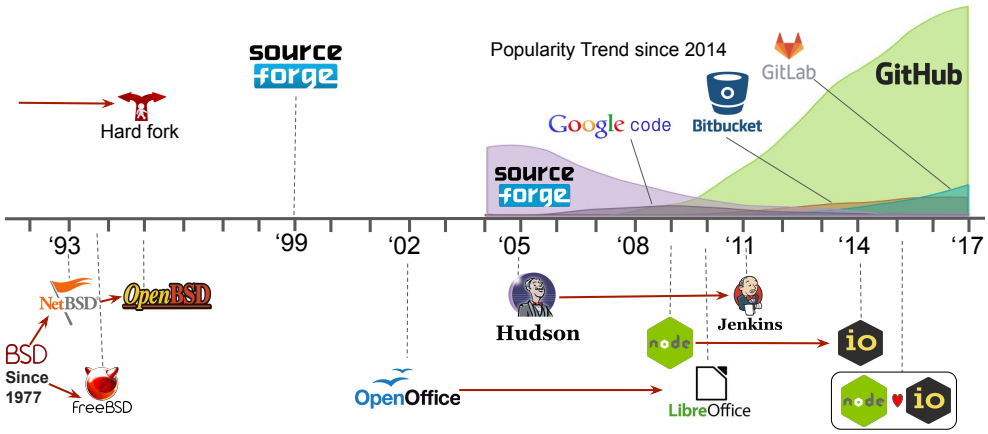


Figure 2.1: Timeline of some popular open-source forking events; popularity approximated with *Google Trends*.

In this thesis, we use the term **(social) fork** in the sense of creating a public copy of a git repository with the intention of integrating those changes back into the original repository and refer to the traditional definition of the splitting of a new independent project as a **hard fork**.

Forking research focused primarily on hard forks in open-source, where a popular topic was understanding the motivations [51, 69, 81, 131, 162, 190, 232], the controversial perceptions around hard forks [42, 85, 131, 161, 181, 237], and the outcomes of hard forks (including studying factors that influence such outcomes) [190, 237]. Specifically, Nyman et al. [162] analyzed self-described reasons for hard forks 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 found that forking can be a suitable practice for variant management [81] and to overcome governance disputes [93].

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 [85, 161, 164], encouraging a survival-of-the-fittest model [234], 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 [85, 131, 161, 181]. Also, Yoo [243] studies the tension between freedom of forking with the challenge of fragmenting the community. There are not many cases where both communities survived after a hard fork, with a prominent but relatively rare example of the BSD variants [163, 179, 180, 190].

However, essentially all that research has been conducted before the rise of social coding, much of it on SourceForge (GITHUB was launched in 2008 and became the dominant open-source hosting site around 2012; cf. Fig 2.1).

In recent years, researchers started studying collaborative development with forks on social coding platforms. The openness of social coding creates transparency [61, 62] 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 [97, 99, 227, 244]. 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.

As the notion of forking has changed over the last 20 yeas, we argue that perceptions and practices around forking could have changed significantly since SourceForge's heydays. In contrast to the strong norm against forking back then, we conjecture that the promotion of social forks on sites like GITHUB, and the often blurry line between social and hard forks, may have encouraged forking and lowered the bar also for hard forks.

Therefore, in this thesis we revisit, replicate, and extend research on hard forks to update and deepen our understanding regarding practices and perceptions around hard forks in order to inform the design of better tools and management strategies to facilitate efficient collaboration.

Also, prior work confirmed that forking provides increased opportunities for community engagement [61, 62, 97, 99, 149]; *e.g.*, over half of the commits in the Marlin project come from forks [214]. 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 [31]. More generally, it has been observed that communities often adopt a shared culture of common practices, but cultures can differ significantly between communities [38].

Overall, most prior works focused on hard forks, though understanding the acceptance of individual contributions through PRs has recently come in focus. In this thesis, we study forks as a contribution mechanism at the project level and focus on factors associated with project-wide inefficiencies.

2.2 Collaboration in Software Engineering Projects

Software engineering projects are inherently collaborative, requiring many software engineers to coordinate their efforts to produce a software system. To ensure the collaboration efficiency, team members are developing shared understanding surrounding multiple artifacts [132, 238]. As the number of people working on a project increases, the potential for communication increases "multiplicatively in proportion to the square of the number of people taking part" [145]. Bandinelli points out that due to the co-operative nature of software development, success is dependent upon "the quality and effectiveness of the communication channels established within the development team" [21]. Thus communication is important.

Research into computer-supported co-operative working (CSCW) suggests a two dimensional model for collaborative work [210]: distance vs. time separation (*i.e.*, same-time same-place, same-time different-place, different-time same-place, or different-time different-place). The work presented in this thesis focuses on examples of *different-time different-place distribution*, and many of the findings are applicable to the other components of the model. The purpose of our study is to form an understanding of collaborative working issues for distributed software teams using fork-based development mechanisms in order to inform future software engineering tools to facilitate efficient collaboration.

Software engineering research has proposed methods aiming for collaboration problems from different perspectives, including documenting and enforcing programmers' intentions [101], physically co-locating development teams to improve communication[105, 220], and improving awareness in the collaborative development environment (see details in Sec. 2.3). This thesis

is focusing on improving awareness for distributed software teams using fork-based development to collaborate by designing new interventions (Chapter 5 and 6).

2.3 Importance of Awareness in Distributed Collaboration

As described in Section 1.1, when the number of forks grows, it becomes *difficult to maintain an overview* of what happens in individual forks and thus of the project's scope and direction. This *lack of an overview* problem is similar to the *lack of awareness* problem that has been studied previously in other distributed software development scenarios [48, 63, 70, 103, 200, 202, 224], but with new challenges. For example, most of the previous works focused on industrial settings, in which all the team members are working on the same project. While in fork-based development, especially in open-source, the number of forks is much bigger than the industry setting, and fork owners are not necessarily contributing to the same projects, which makes creating an overview for the community and identifying the useful information for individual developer even harder. Therefore it is necessary to design awareness tool under the fork-based development environment.

Dourish et al. defined awareness as "an understanding of the activities of others, which provides a context for your own activity", and demonstrate that awareness of individual and group activities are critical to successful collaboration because it helps group members to better understand of sequence and timing of things and the temporal boundaries of their actions [70]. In the software development environment, awareness means that team members can become aware of the work of others that is interdependent with their current tasks, therefore enabling better coordination of teams.

Developers may be globally distributed [171] while collaborating on the same project. Distance creates an additional challenge to software development processes, because of fewer opportunities for rich interaction and lower frequencies of direct communication [108]. Especially, for an open-source project, it is common to require distributed software developers to coordinate their efforts [156]. The distance affects collaboration issues, such as awareness and communication [109, 198]. Thus, it is important to keep awareness of the activity of the projects and other developers in such a distributed collaboration situation [102, 103].

There are many kinds of information need to be aware of, such as the technical and social aspects of the development [65], current and upcoming articulation work [151], the overall status of their projects and critical deadlines, understand current priorities and bottlenecks, dependencies between components and teams, and need to be informed of changes to tasks they are working on in a real-time manner [224]. For example, there are awareness tools for software development to focus on low-level code-specific tasks [204], like seesoft [76] and Augur [89].

Gutwin et al. [103] defined workspace awareness as *the up-to-the-moment understanding of another person's interaction with the shared workspace*, which means an understanding of actions on shared artifacts. Correspondingly, researchers have investigated different approaches to provide workspace awareness [22, 89, 166, 201]. For instance, FastDASH [32] shows when two developers are working concurrently on a file, allowing them to preempt merge conflicts; Palantír [202] follows a similar approach by notifying developers when changes are made to relevant work artifacts. In addition, there are tools presenting the high-level activities, like

project management issues, change requests and social and historical patterns in the development process [78]. For example, the World View in Palantír also addresses "awareness in the large" [200], which provides a comprehensive view of the team dynamics of a project especially for the geographical location of developers. Furthermore, practitioners designed toolsets targeting distributed and collaborative software development, such as IBM's Jazz [90], Microsoft's CollabVS [104].

Workspace awareness is also important for open-source software development. Developers need to seek out information about their fellow developers in order to stay aware of their work activities [226]. Specifically, people seek awareness information such as who is working on what part of the project from simple text communication such as mailing lists and text chat to stay aware of the work of other developers on the project [103]. Newcomers need to seek similar awareness information from text communication tools in order to "recruit" core project developers towards supporting their contributions [73]. Also, developers on a project used similar work awareness information from the mailing list in order to select code contributions to review.

Gutwin et al. [102] present techniques that address the *visibility problem*, when the workspace is larger than a member's screen and when people can move their views independently. The paper shows that overview representation that shows the entire workspace in miniature and provides a high-level perspective on artifacts and events in unseen areas of the workspace, is the most useful view to help people to maintain workspace awareness. Because of visual awareness information makes it easier to communicate useful information without talking, and awareness information gives people confirmation about the other person's activities [102].

However, too much awareness could be a problem leading to information overload [187]. In our project, we also found that developers think it is hard to quickly get an overview of the community, except checking each fork one by one, although, with the advent of transparent development environments, all the information is publicly available [61]. Communication happens when transparency break down – there was certain information developers could not directly observe [61]. Thus, there is an opportunity for extracting useful information and summarizing the activity of each fork to provide an overview.

Furthermore, an uncoordinated team – with lack of communication – tends to lose the notion of who is changing which parts of the system (awareness), which leads to merging conflicts and duplicated work [63]. In our work, we also found inefficiencies, such as redundant development, lost contribution (described in Chapter 1), happened in fork-based development mechanism because of a lack of awareness.

Thus, in order to help globally distributed developers to gain a better overview, better communicate, and find the potential collaborator, we will design awareness tools in fork-based development to mitigate inefficiencies because of lacking awareness, such as summarizing information of un-merged code changes information in forks, and detecting potentially redundant development (see details in Chapter 5 and 6).

Chapter 3

Identifying Natural Interventions from Best Practices

This chapter shares material with the FSE'19 paper "What the Fork: A Study of Inefficient and Efficient Forking Practices in Social Coding" [250].

As we discussed in Chapter 1, modern tools and platforms (*e.g.*,GITHUB, Bitbucket, and GITLAB) have made forking easier (1) to track and integrate changes across multiple forks without central management and (2) to publish changes, including incomplete and experimental ones. Forking has become very common and popular [97, 179].

While easy to use and popular in practice, fork-based development has well-known downsides. In this dissertation, we study one of the problems – a lack of an overview and corresponding inefficiencies: *lost contribution, redundant development, rejected pull requests, and fragmented communities.* In this chapter, 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 efficient forking practices?** Understanding what influences inefficiencies correlated with higher collaboration efficiency.

Concretely, we *derived potential characteristics and practices* that could affect forking (in) efficiency by (1) asking open-source developers about their forking practices and (2) exploring exiting theories on distributed collaboration. We then designed a *cross-sectional correlational study* to test these hypotheses at scale on GITHUB data (study overview is shown in Figure 3.1). Specifically, we *designed measures* for four inefficiencies and potential characteristics and practices, collected data from 1311 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

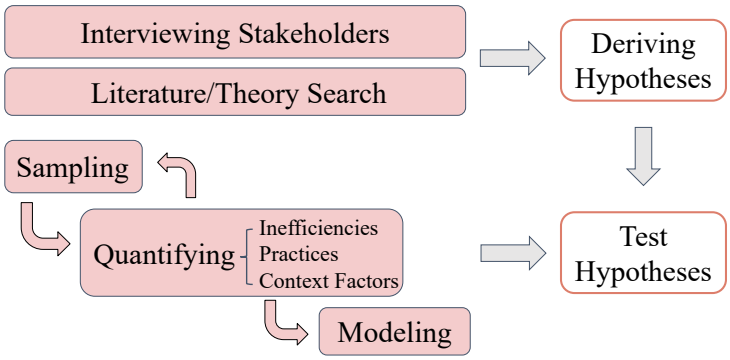


Figure 3.1: Outline of studying natural intervention.

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 [40].

In a nutshell, here we study the **natural interventions** that are correlated with efficient forking practices, and identify the best practices that are potentially helpful to improve collaboration efficiencies for other projects.

3.1 Identifying Potential Context Factors and Deriving Hypotheses

To identify potential context factors of a project that are potentially correlated with forking efficiency, we pursued two strategies in parallel: interviews with active open-source contributors and analysis of the literature on distributed collaboration (shown in Figure 3.1). 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 pull requests, 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.

3.1.1 Modularity affects forking practices

Interviews. Discussions with contributors familiar with both *Marlin* and *Smoothieware* revealed an interesting contrast: *Marlin* and *Smoothieware* are both frequently forked open-source firmware projects for 3D printers (*Marlin* has over 8,800 forks on GITHUB and *Smoothieware* has over 821 forks), but contributors perceive practices in both projects as very different. Learning from *Marlin*'s maintenance problem 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 implementation. 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. Another developer who has developed significant *Marlin* extensions in a fork without attempting to merge them back mentioned that one of the reasons for not merging is that *Marlin*'s structure causes high integration effort. Interestingly, some GITHUB projects have an extremely modular structure, *e.g.*, a collection of scripts or plug-ins that are assembled automatically (such as *homebrew* package descriptions), 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, Conway's law postulated that structure of the code mirrors the structure of the organization [55]. Parnas defined a module as "a responsibility assignment rather than a subprogram", which indicates that dividing a software system is simultaneously a division of labor [169]. In addition, Herbsleb et al. revisited Conway's law by conducting a user study and found that integration turned out to be the most difficult part of a geographically distributed software project [107]. The result shows that in order to reduce the need for cross-site communication as much as possible, it is better to assign work to different sites according to the architectural separation in a design that is as modular as possible, and only split the development of well-understood products (or parts of products), where plans, processes, and interfaces are established and likely to be very stable.

Researchers and practitioners have also emphasized the importance of modularity for open-source development [128, 152, 222]. For example, Torvalds [222] claims *"for without [modularity], you cannot have people working in parallel"* and Midha and Palvia [152] found that modularity is positively associated to technical success of open-source projects. Specifically, MacCormack et al. [143] suggested that more modular projects could be more attractive to potential contributors. Similarly, Baldwin et al. found that a greater number of modules can yield more design options, which creates more opportunities for the exchange of valuable work among developers and increases developers' incentives to work on the codebase [19]. 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.

In summary, modularity is important for both technical and organizational perspectives, so we suspect modularity as a collaboration mechanism could be also a good practice for fork-based development. This aligns with our hypothesis that projects with a better modular design have a larger portion of contributing forks.

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.*

3.1.2 Coordination mechanisms affect forking practices

Interviews. Interviewees of many projects, including *Marlin* and *Smoothieware*, indicated that their communities welcome all pull requests 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* (25,200 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 [140]). The issue tracker records which forks contain the corresponding code changes for each issue; other forks are of little interest to maintainers and unsolicited pull requests remain ignored for years. Similarly, one of the maintainers of the Python machine-learning project *scikit-learn* (19,300 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 [5]. A maintainer of *scikit-learn* was even surprised about the existence of duplicate pull requests, 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 et al. [41] 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. [176] 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 [205] 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.

As more design options justify multiple efforts directed at the same target, implicitly create tournaments in which developers can compete to provide the best design, which developers may intentionally duplicate each other's efforts in order to obtain a higher best outcome [19]. To mitigate this problem, Baldwin et al. studied that different programmers can communicate to avoid redundant work. Since workers share costs, a collective effort with adequate communication is always preferable to coding in isolation.

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 pull requests because fewer pull requests misalign with the maintainer's vision. We also hypothesize that *pre-communication*, *i.e.*, developers discussing their contributions before submitting pull requests, associates with fewer redundant pull requests:

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

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

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.*

3.1.3 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 pull request 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 2.1, 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 [18, 160].

Also, researchers found that rejected contributions demotivate developers and discouraging them from submitting contributions in the future [212]. Then developers decide to continue on their own, rather than merging their changes this may fragmented communities, which hurts the sustainability of a community. Thus, we explore related work about understanding the trade-offs between different management strategies.

Meanwhile, researchers have found that many aspects of a software system are difficult to implement modularly [219] and that too rigid compatibility requirements might hinder innovation [38]; also, modularity does not always align with how developers think [167].

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 pull request 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 pull requests could at the same time reduce efficiency regarding community fragmentation.

3.1.4 Summary

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 (see Figure 3.2), 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. 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 [26, 66].

3.2 Operationalization

To quantitatively test our hypotheses, we subsequently operationalize measures for context factors and inefficiencies, collect data from 1131 GitHub repositories. Specifically, we iteratively developed outcome **measures for the inefficiencies** (lost contribution, redundant development,

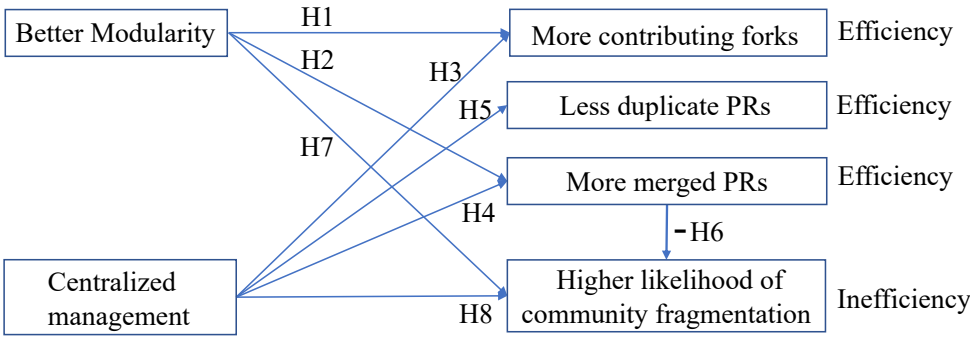
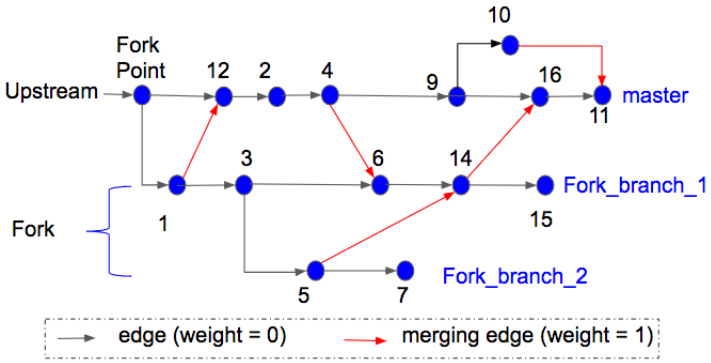
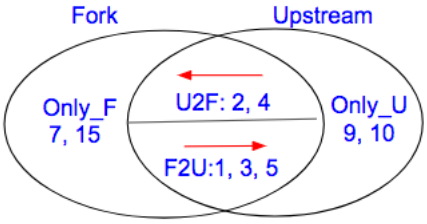


Figure 3.2: Eight Hypothesis of characteristics and practices that could affect forking (in) efficiency.

rejected pull requests, and fragmented communities), measures for **context factors** (modularity, coordination mechanisms, and contribution barriers), and measures for **control variables**. 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 [13].



(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 3.3: Determining the origin of commits.

3.2.1 Outcome: Ratio of contributing forks.

To assess inefficiencies regarding lost contributions (see Section 1.1), we measure the fraction of active forks in which developers have submitted pull requests or otherwise integrated their code changes into the upstream project (higher values indicate higher efficiency). Specifically, we query the GITHUB API to identify whether pull requests 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 pull requests.

Unfortunately, reliably detecting active forks and merged changes is not trivial. Forks may pull changes from upstream, upstream repositories can merge changes also without pull requests, 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 3.3(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 [31], 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 pull requests originating from the fork.

3.2.2 Outcome: Ratio of merged pull requests.

To assess inefficiencies regarding rejected pull requests (see Section 1.1), we measure the fraction of closed pull requests that have been accepted (higher values indicate more efficient outcomes). The resolution status reported by GITHUB is often not reliable [97], as many developers integrate

pull requests through other mechanisms than GITHUB's user interface, thus closing them without marking them as accepted. We follow Gousios' heuristics [97] to identify accepted contributions, but refine them to account for frequent practices we observed:

- If the pull request is marked as merged on GITHUB, we mark it as accepted. (83.2% of all merged pull requests).
- If a commit closes the pull request (using certain phrase conventions advocated by GITHUB, *e.g.*, `fixes #1234`) and that commit appears in the target project's branch, we consider the pull request as accepted. Different from Gousios' work, we use GITHUB's issue events timeline API, rather than analyzing textual comments, to detect links to pull requests in commit messages. (8.8% of all merged pull requests).
- If any of the last 3 discussion comments of the pull request refers to a commit SHA, we consider the pull request 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 pull request appears in the comment, indicating a competing or superseding PR. (0.15% of all merged pull requests).
- If the last comment before closing the pull request matches both rules (1) and (2) above, or matches only rule (3), we consider the pull request as accepted, unless a link to another pull request appears in the comment. (7.9% of all merged pull requests).

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

3.2.3 Outcome: Ratio of duplicate pull requests.

To assess inefficiencies regarding duplicate development (see Section 1.1), we measure the fraction of closed pull requests rejected due to redundant work (lower values indicate higher efficiency). To identify duplicate pull requests, we refined heuristics, summarized and validated by Yu et al. [245], based on regular expressions to identify duplicate-related keywords in pull request comments and links to other pull requests. We also found many cases in which a pull request 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 pull requests rejected due to redundant development that can be found in the implementation [13].

3.2.4 Outcome: Presence of hard forks.

To assess inefficiencies regarding community fragmentation, we measure whether projects have at least one hard fork (see Section 1.1). We consider a fork as a hard fork if (a) it has attracted its own external contributions (at least two pull requests submitted by other contributors) or (b) it has substantial unmerged changes (at least 100 commits, as identified from our commit graph, see Figure 3.3) 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.

3.2.5 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 [33, 79]; others measure logic coupling from co-change patterns observed in the project's revision history [25, 45, 252]. To measure modularity uniformly across projects in different programming languages, we adopt a light-weight previously validated measurement of logic coupling, *ROSE* [252]: 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.

3.2.6 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.

3.2.7 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 PR) 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.

3.2.8 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 [6], developers might submit an incomplete pull request clearly marked as 'work in progress' (*e.g.*, using labels) and later update that pull request 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 pull requests for which the author has commented under the linked issue *before* submitting the pull request or in which the pull request was marked as work in progress in its history out of all closed pull requests 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.

3.2.9 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 pull requests by *external* (non-core) contributors, modeled closely after factors that prior research found to correlate with the chance of accepting individual pull requests [97, 227]: (1) SubmitterPriorExperience – a dummy encoding whether at least half of the pull requests in the project are submitted by people with prior experience submitting and having merged pull requests in the same project in the past; pull requests from people with prior experience are more likely to be accepted [97]. (2) Ratiopull requestsWithTests – the ratio of pull requests containing test cases; pull requests containing test cases are more likely to be accepted [97]. We reused our measure to identify tests [225] 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 pull requests of the number of commits on files touched by each pull request during the previous three months prior to the pull request creation; pull requests touching "hot" files, changed frequently in the recent past, are more likely to be accepted [97]. and (4) SubmitterSocialConnections – a dummy encoding whether at least half of the pull requests in the project are submitted by people who followed (already at pull request creation time) the maintainer who closed each respective PR; pull requests by more socially connected submitters, who follow the maintainers, are more likely to be accepted [227].

3.3 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 [96] dump, we filtered projects based on the a list of 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.

Table 3.1: How we stratified our sample.

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

- *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 only 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 3.1; 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 1311 projects for our analysis.

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

Since computing the flow graph in Figure 3.3 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 pull requests.

In Figure 3.4, 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.

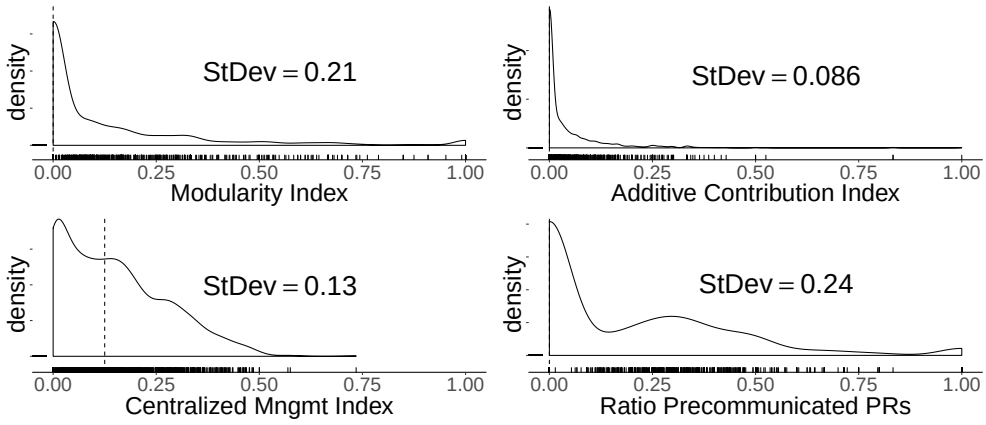


Figure 3.4: Density plots for our main predictors. The dashed line denotes the median.

3.4 Statistical Analysis

We use multiple regression modeling to test, for each outcome, whether it is significantly associated with the different hypothesized context factors, while controlling for known confounding variables, cf. prior work. The multivariate nature of our analysis is especially relevant when modeling the pull request merge ratio, which is known to be impacted by the presence of tests and the prior experience of the pull request submitters [97, 227].

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 3.6 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).

3.5 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. In addition, the history of Git repositories is not reliable, as users can rewrite histories after the fact, and merges are difficult to track if code changes are merged as a new commit or through 'squashing' and 'rebasing' rather than through a traditional merge commit. As a consequence, despite best efforts, there will be inaccuracies in our operationalization of ratio of contributing forks, which we expect will lead to some mis-

classification of merged code.

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 [13].

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 [126], they likely do not share the same context of loosely-coordinated distributed contributions from developers outside a core team.

3.6 Result

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

Table 3.2: 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	

3.6.1 When do forks attempt to contribute back? ($\mathbf{H}_1$, $\mathbf{H}_3$)

To test our hypotheses that modularity ($\mathbf{H}_1$) and coordination practices ($\mathbf{H}_3$) 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 3.2, 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 $\mathbf{H}_1$ and $\mathbf{H}_3$.

Table 3.3: 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	

3.6.2 When are more contributions integrated? ($\mathbf{H}_2$, $\mathbf{H}_4$)

To test our hypotheses whether modularity ($\mathbf{H}_2$) and coordination mechanisms ($\mathbf{H}_4$) 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 experience of the pull request submitters, the ratio of pull requests containing test cases, and the median pull request hotness.

In Table 3.3, 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 $\mathbf{H}_4$.

3.6.3 When is duplicate work more common? ($\mathbf{H}_5$)

To test whether discussing or claiming an issue before submitting a pull request 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).

Table 3.4: 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	

The regression summary in Table 3.4 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.

3.6.4 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.

Our model, summarized in Table 3.5, 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 ($\mathbf{H}_7$).

3.7 Discussion

3.7.1 Modularity

Modularity has been widely recognized as an important quality that facilitates software evolution and eases division of labor and collaboration [20, 55, 150, 169]. Our study confirms that better modularity is associated with higher efficiency of distributed fork-based development,

Table 3.5: 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	

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 pull request acceptance ratios ($\mathbf{H}_2$); in turn, higher pull request 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 pull request 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.

3.7.2 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 [61], 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.

3.7.3 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 [136, 185] might be an interesting alternative strategy.

3.8 Implications

3.8.1 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.

3.8.2 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 [10, 11, 12, 184, 249] 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 [8], as have community tooling for coordination [10], which would be worth evaluating. There may be research opportunities to detect redundant pull requests automatically to reduce the maintainers' effort [136, 185, 245] or even to detect redundant development early before developers finished their work [185]. Research on mentoring [47, 82] 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* [225] or *metric dashboards* [40, 46] 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 [71], and recently several researchers have explored lightweight tooling to support fork-based variant management [17, 84, 211].

3.9 Summary

In this chapter, we investigated the research question: **What characteristics and practices of a project associate with efficient forking practices?** Specifically, we interviewed stakeholders and conducted literature search to derive eight hypotheses about project characteristics and practices that could affect forking (in) efficiency. We then designed cross-sectional correlational study to test the hypotheses. 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. 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. Moreover, 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. This is one of the solutions that we studied to mitigate collaboration inefficiencies when using fork-based development mechanisms, which is identifying natural interventions. And this is a complementary solution to the new interventions that we designed and described in Chapter 5 and 6.

Chapter 4

A Study of Hard Forks on GitHub

This chapter shares material with the ICSE'20 paper "How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub" [251].

As discussed in Chapter 2.1, the common notion of a fork has changed: Traditionally, forking was the practice of copying a project and splitting off new *independent* development, and often intended to compete with or supersede the original project [85, 131, 161, 181]. Nowadays, forks are typically understood to be 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. In this dissertation, we define the former as **hard fork**, which would leads to the inefficiency of fragmented community; and we define the latter as **(social) fork**.

In chapter 3, we identified hard forks by the name change, number of unmerged commits, or whether the fork is receiving external pull requests over 1311 projects, and studied the community fragmentation phenomena as an indicator of collaboration inefficiency. In this chapter, we take one step further to study hard forks over a larger number of GITHUB projects.

Since we observe the change of the notion of forking, we argue that perceptions and practices around forking could have changed significantly since SourceForge's heydays. In contrast to the strong norm against forking back then, we conjecture that the promotion of social forks on sites like GITHUB, and the often blurry line between social and hard forks, may have encouraged forking and lowered the bar also for hard forks. Therefore, in this chapter, we update and deepen our understanding regarding practices and perceptions around hard forks can inform the design of better tools (see chapter 5 and 6) and management strategies to facilitate efficient collaboration (see chapter 3).

4.1 Motivation

Prior research into forking of free and open-source projects focused on the motivations behind hard forks [51, 69, 81, 131, 162, 190, 232], the controversial perceptions around hard forks [42, 85, 131, 161, 181, 237], and the outcomes of hard forks (including studying factors that influence such outcomes) [190, 237]. However, essentially all that research has been conducted before the

rise of social coding, much of it on SourceForge (GITHUB was launched in 2008 and became the dominant open-source hosting site around 2012; cf. Figure 2.1). Therefore, we argue that it is time to revisit, replicate, and extend research on hard forks, asking the central question of this work: **How have perceptions and practices around hard forks changed?**

In this chapter, we describe a mixed-methods empirical design, combining repository mining with 18 developer interviews, the goal is to **further investigate the frequency, common evolution patterns, and perceptions of hard forks in the current social coding environment.**

In this chapter, we investigate:

- **Frequency of hard forks:** We attempt to quantify the frequency of hard forks among all the (mostly social) forks on GITHUB. Specifically, we design and refine a classifier to automatically detect hard forks. We find 15,306 instances, showing that hard forks are a significant concern, even though their relative numbers are low.
- **Common evolution patterns of hard forks:** We classify the evolution of hard forks and their corresponding upstream repository to observe outcomes, including whether the fork and upstream repositories both sustain their activities and whether they synchronize their development. We develop our classification by visualizing and qualitatively analyzing evolution patterns (using card sorting) and subsequently automate the classification process to analyze all detected hard forks. We find that many hard forks are sustained for extended periods and a substantial number of hard forks still at least occasionally exchange commits with the upstream repository.
- **Perceptions of hard forks:** In interviews with 18 open-source maintainers of forks and corresponding upstream repositories, we solicit practices and perceptions regarding hard forks and analyze whether those align with ones reported in pre-social-coding research. We find that the 'stigma' often reported around hard forks is largely gone, indeed forks including hard forks are generally seen as a positive, with many hard forks complementing rather than competing with the upstream repository. Futhermore, with social forking encouraging forks as contribution mechanism, we find that many hard forks are not deliberately planned but evolve slowly from social forks.

Overall, we contribute (1) a method to identify hard forks, (2) a dataset of 15,306 hard forks on GITHUB, (3) a classification and analysis of evolution patterns of hard forks, and (4) results from interviews with 18 open source developers about the reasons for hard forks, interactions across forks, and perceptions of hard forks.

Our research focuses on development practices on GITHUB, which is by far the dominant open-source hosting platform (cf. Figure 2.1) and has been key in establishing the social forking phenomenon. Even large projects primarily hosted on other sites often have a public mirror on GITHUB, allowing us to gather a fairly representative picture of the entire open-source community. Our main research instruments are semi-structured interviews with open-ended questions and repository mining with GHTORRENT [98] and the GITHUB API. While our research is not planned as an exact replication of prior work and exceeds the scope of prior studies by comparing social and hard forks, many facets seek to replicate prior findings (*e.g.*, regarding motivations and outcomes of hard forks) and can be considered a conceptual replication [125, 203].

4.2 Research Questions and Methods

As described in Sec. 2.1, the conventional use of the term *forking* as well as corresponding tooling have changed with the rise of distributed version control and social coding platforms, and we conjecture that this also influenced hard forks. Hence, our overall research question is **How have perceptions and practices around hard forks changed?**

We explore different facets of hard forks, including motivations, outcomes, and perceived stigma (cf. Sec. 2.1). We also attempt to identify how frequent hard forks are across GITHUB, and discuss how developers navigate the tension and often blurry line between social and hard forks. We adopt a concurrent mixed-method *exploratory* research strategy [57], in which we combine *repository mining* – to identify hard forks and their outcomes – with *interviews* of maintainers of both forks and upstream projects – to explore motivations and perceptions. Mixing multiple methods allows us to explore the research question simultaneously from multiple facets and to triangulate some results. In addition, we use some results of repository mining to guide the selection of interviewees.

We explicitly decided against an *exact replication* [125, 203] of prior work, because contexts have changed significantly. Instead, we guide our research by previously explored facets of hard forks, revisit those as part of our repository mining and interviews, and contrast our findings with those reported in pre-GITHUB studies. In addition, we do not limit our research to previously explored facets, but explicitly explore new facets, such as the tension between social and hard forks, that have emerged from technology changes or that we discovered in our interviews.

4.2.1 Instrument for Visualizing Fork Activities

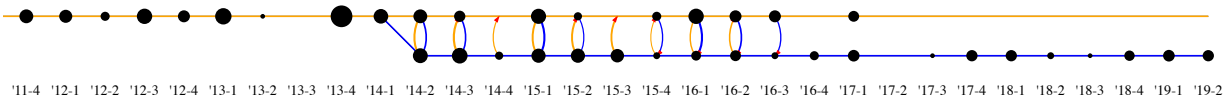


Figure 4.1: An example of commit history graph of fork `tmyroadctfig/jnode`

We created *commit history graphs*, a custom visualization of commit activities in forks, as illustrated in Figure 4.1, to help develop and debug our classifiers (Sec. 4.2.2 and 4.2.3), but also to prepare for interviews. Given a pair of a fork and corresponding upstream repositories, we clone both and analyze the joint commit graph between the two, assigning every commit two one of five states: (1) *created before the forking point*, (2) *only upstream* (not synchronized), (3) *only in fork* (unmerged), (4) *created upstream but synchronized to the fork*, and (5) *created in the fork but merged into upstream*. Technically, in a nutshell, we build on our prior commit graph analysis [250], where merge edges are assigned weight 1 and all other edges weight 0, and the shortest path from the commit to any branch in either fork or upstream repository identifies where the commit originates and whether it has been merged (and in which direction).[1]

[1]There are a few nuances in the process due to technicalities of Git and GITHUB. For example, if the upstream repository deletes a branch after forking, the joint commit graph would identify the code as exclusive to the fork; to that end, we discard commits that are older than the forking timestamp on GITHUB. Such details are available in our open-source implementation (`https://github.com/shuiblue/VisualHardFork`).

We subsequently plot activities in the two repositories over time, aggregated in three-month intervals; larger dots indicate more commits. In these plots, we include additional arrows for synchronization (from upstream into the fork) and merge (from fork to upstream) activities. With these plots, we can quickly visually inspect development activities before and after the forking point as well whether the fork and the upstream repository interact.

4.2.2 Identifying Hard Forks

Identifying hard forks reliably is challenging. Pre-GITHUB work often used keyword searches in project descriptions, *e.g.*, 'software fork', or relied on external curated sources (*e.g.*, Wikipedia) [190]. Today, on sites like GITHUB, hard forks use the same mechanisms as social forks without any explicit distinction.

Classifier development. For this work, we want to gather a large set of hard forks and even approximate the frequency of hard forks among all 47 million forks on GITHUB. To that end, we need a scalable, automated classifier. We are not aware of any existing classifier except our own prior work [250], in which we classified forks as hard forks if they have at least two own pull requests or at least 100 own, unmerged commits and the project's name has been changed. Unfortunately, we found that this classifier missed many actual hard forks (false negatives), thus we went back to the drawing board to develop a new one.

We proceeded iteratively, repeatedly trying, validating, and combining various heuristics. That is, we would try a heuristic to detect hard forks and manually sample a significant number of classified forks to identify false positives and false negatives, revising the heuristic or combining it with other steps. Commit history graphs (cf. Sec. 4.2.1) and our qualitative analysis of forks (Sec 4.2.3 below) were useful debugging devices in the process. We iterated until we reached confidence in the results and a low rate of false positives.

Our final classifier proceeds in two steps: first, we use multiple simple heuristics to identify *candidate* hard forks; second, we use a more detailed and more expensive analysis to decide which of those candidates are *actual* hard forks.

In the **first step**, we identify as candidate hard forks, among all repositories labeled as forks on GITHUB, those that:

- *Contain the phrase "fork of" in their description* (H_1). We use GITHUB's search API to find all repositories that contain the phrase "fork of" in their project description and are a fork of another project. The idea, inspired by prior work [162], is to look for projects that explicitly label themselves as forks (defined as "*self-proclaimed forks*"), *i.e.*, developers explicitly change their description after cloning the upstream repository. To work around GITHUB's API search limit of 1000 results per query, we partitioned the query based on different time ranges in which the repository was created. Next, we compare the description of the fork and its upstream project to make sure the description is not copied from the upstream, i.e., that the upstream project is not already a self-proclaimed fork.
- *Received external pull requests* (H_2). Using the June 2019 GITHUB dataset [96], we identified all GITHUB repositories that are labeled as forks and have received at least *three* pull requests (excluding pull requests issued by the fork's owner to avoid counting developers

who use a process with feature branches). We consider external contributions to a fork as a signal that the fork may have attracted its own community.

- *Have substantial unmerged changes* (H_3). Using the same GHTORRENT dataset, we identify all forks that have at least 100 own commits, indicating significant development activities beyond what is typical for social forks.
- *Have at least 1-year of development activity* (H_4). Similar to the previous heuristic, we look for prolonged development activities beyond what is common for social forks. Specifically, we identify those forks as candidates in which the time between the first and the last commit spans more than one year.
- *Have changed their name* (H_5). We check if the fork's name in GITHUB has been changed from the upstream repository's name (with Levenshtein distance ≥ 3). This heuristic comes from the observation that most social forks do not change names, but that forks intending to go in a different direction and create a separate community tend to change names more commonly (*e.g.*, *Jenkins* forked *Hudson*).

Each repository that meets *at least one* of these criteria is considered as a candidate. We show how many candidates each heuristic identified in the second column of Figure 4.2c. Note, for all heuristics that use GHTORRENT, we additionally validated the results by checking whether the fork and upstream pair still exist on GITHUB and whether the measures align with those reported by the GITHUB API.[2]

In the **second step**, we performed more detailed (and expensive) analyses of commit graphs and repository metadata in each candidate hard fork, to filter false positives (details of the filtering criteria in the paper [251]).

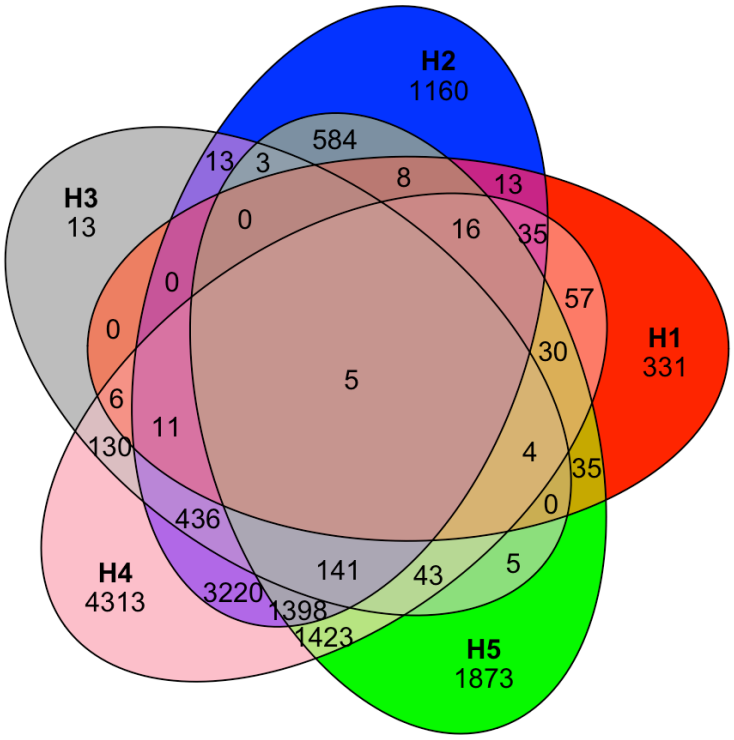
Classifier validation. To validate the precision of our classifier, we manually inspected a random sample of 300 detected hard forks. By manually analyzing the fork's and the upstream repository's history and commit messages, we classified 14 detected hard forks as likely false positives, suggesting an acceptable accuracy of 95 %. Note that manual labeling is a best effort approach as well, as the distinction between social and hard fork is not always clear (see also our discussion of interview results in Sec. 4.3.4).

Analyzing false negatives (recall) is challenging, because hard forks are rare, projects listed in previous papers are too old to detect in our GitHub dataset, and we are not aware of any other labeled dataset. We have manually curated a list of known hard forks from mentions in web resources and from mentions during our interviews. Of the 3 hard forks of which both the fork and the upstream repository are on GitHub, we detect all with our classifier, but the size of our labeled dataset is too small to make meaningful inferences about recall.

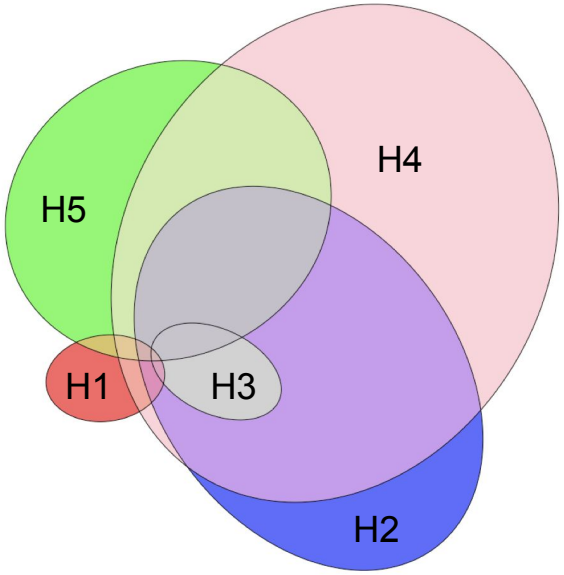
4.2.3 Classifying Evolution Patterns

We identified different evolution patterns among the analyzed forks using an iterative approach inspired by card sorting [209]. Evolution patterns describe how a hard fork and the corresponding

[2]We include this step after identifying occasional errors in GHTORRENT in our validation steps, such as switched fork-upstream relations between two repositories.



(a) Overlap between the heuristics (with detailed intersections).



(b) Overlap between the heuristics (Proportional).

Rule	Candidates	Actual
H_1	10,609	551
H_2	23,109	7,043
H_3	14,956	810
H_4	33,073	11,268
H_5	20,358	5,568
Total	63,314	15,306

(c) Hard forks identified.

Figure 4.2: Statistics on identified candidate hard forks and actual hard forks.

upstream project coevolve and can help to characterize forking outcomes. In addition, we used evolution patterns to diversify interviewees.

Specifically, we printed cards with commit history graphs of 100 randomly selected hard forks (see Sec. 4.2.2), then all three authors jointly grouped the cards and identified a few common patterns. Our card-sorting was open, meaning we had no predefined groups; the groups emerged and evolved during the analysis process. Afterward, we manually built a classifier that detects the forks for each identified pattern. We then applied this classifier to the entire dataset, inspected that the automatically classified forks actually fit the patterns as intended (refining the classifier and its thresholds if needed). We then picked another 100 hard forks that fit none of the previously defined patterns and sorted those again, looking for additional patterns. We similarly proceeded within each pattern, looking at 100 hard forks to see whether we can further split the pattern. We repeated this process until we could not identify any further patterns.

After several iterations, we arrived at a stable list of 15 patterns with which we could classify 97.7 % of all hard forks. We list all patterns with a corresponding example commit history graph in Table 4.2. The patterns use characteristics that relate to previously found outcomes, such as fork or upstream being discontinued, but also consider additional characteristics corresponding to features that were not available or easily observable before distributed version control, *e.g.*, whether the fork and upstream merge or synchronize. We present the patterns in a hierarchical form, because our process revealed a classification with a fairly obvious tree structure, not because we were specifically looking for a hierarchical structure.

4.2.4 Interviews

To solicit views and perceptions, we conducted 18 semi-structured interviews with developers, typically 20 to 40 minutes. Despite reaching fewer developers, we opted for interviews rather than surveys due to the exploratory nature of our research: Interviews allow more in-depth exploration of emerging themes.

Interview protocol. We designed a protocol [15] that covers the relevant dimensions from earlier research and touches on expected changes, including reasons for forking, perceived stigma of forking, and the distinction and possible tensions between social and hard forks. We asked *fork owners* about their decision process that lead to the hard fork, their practices afterward (*e.g.*, why they renamed the projects), their current relationship to the upstream project (*e.g.*, whether they still monitor or even synchronize), and their future plans. In contrast, we asked *owners of upstream projects* to what extent they are aware of, interact with, or monitor hard forks; and to what degree they are concerned about such forks or even take steps to avoid them. In addition, we asked all participants with a long history of open-source activity if they observed any changes in their practices or perceptions and that of others over time.

All interviews were semi-structured, allowing for exploration of topics that were brought up by the participants. Our interview protocol evolved with each interview, as we reacted to confusion about questions and insights found in earlier interviews. That is, we refined and added questions to explore new insights in more detail in subsequent interviews – for example, after the first few interviews we added questions about the tradeoff between being inclusive to changes versus

Table 4.1: Background information of participants.

Par.	Domain	#Stars(U)	#Stars(F)	LOC	Role	Exp.(yr)
P1	Blockchain	<20	<10	10K	F	19
P2	Reinforcement learning	10K	1K	30K	F	3
P3	Mobile processing	-	70	20K	F	6
P4	Video recording	-	100	300K	F	18
P5	Helpdesk system	2K	<10>	800K	F	5
P6	CRM system	30	200	800K	F	10
P7	Physics engine	-	300	100K	F	15
P8	Social platform	500	230	500K	F	20
P9	Reinforcement learning	<20	<20	30K	2nd-F	3
P10	Game Engine	500	<10	200K	2nd-F	21
P11	Networking	300	100	500K	F	10
P12	Email library	-	10K	20K	F/U	32
P13	Game engine	3K	70	20K	F	11
P14	Machine learning	30K	50	60K	F	8
P15	Image editing	70	<10	20K	F	20
P16	Image editing	70	<10	20K	U	10
P17	Microcontrollers	9K	1K	300K	U	6
P18	Maps	400	<10	100K	U	9

F: Hard Fork Owner; U: Upstream Maintainer; 2nd-F: Fork of the Hard Fork
*Some of the upstream projects are not in GITHUB,
so the number of stars is unknown. Numbers rounded to one significant digit.

risking hard forks and questions regarding practices and tooling to coordinate across repositories. To ground each interview in concrete experience rather than vague generalizations, we focused each interview on a single repository in which the interviewee was involved, bringing questions back to that specific repository if the discussion became too generic.

Participant recruitment. We selected potential interviewees among the maintainers of the 15,306 identified hard forks and corresponding upstream repositories. We did consider maintainers with public email address on their GITHUB profile that were active in the analyzed repositories within the last 2 years (to reduce the risk of misremembering). We sampled candidates from all evolution patterns (Sec. 4.2.3) and sent out 242 invitation emails.[3]

Overall, 18 maintainers volunteered to participate in our study (7 % response rate). Ten opted to be interviewed over email, one through a chat app, and all others over phone or teleconferencing. In Table 4.2, we map our interviewees to the evolution pattern for the primary fork discussed (though interviewees may have multiple roles in different projects). Naturally, our interviewees are biased toward hard forks that are still active. Our response rate was also lower among maintainers of upstream repositories, who were maybe less invested in talking about forking. In Table 4.1, we list information about our interviewees and the primary hard fork we discussed. All interviewees are experienced open-source developers, specifically, many with more than 10 years experience of participating in open-source community, meaning they have interacted with

[3]We unfortunately could not recruit interviewees in all roles for all patterns. For example, for 'reviving a dead project' we would not find any upstream maintainers that were active in the last 2 years.

earlier open-source platform such as *Sourceforge*. Our interviews reached saturation, in that the last interviews provided only marginal additional insights.

Analysis. We analyzed the interviews using standard qualitative research methods [196]. After transcribing all interviews, two authors coded the interviews independently, then all authors subsequently discussed emerging topics and trends. Questions and disagreements were discussed and resolved together, if needed asking follow up questions to some interviewees.

4.2.5 Threats to Validity and Credibility

Our study exhibits the threats to validity and credibility that are typical and expected of this kind of exploratory interview studies and the used analysis of archival GitHub data.

Distinguishing between social and hard forks is difficult, even for human raters, as the distinction is primarily one of intention. In our experience, we can make a judgment call with high inter-rater reliability for most forks, but there are always some repositories that cannot be accurately classified without additional information. We build and evaluate our classifiers based on a best effort strategy, as discussed.

While we check later steps with data from the GITHUB API, early steps to identify candidate hard forks may be affected by missing or incorrect data in the GHTorrent dataset. In addition, the history of Git repositories is not reliable, as timestamps may be incorrect and users can rewrite histories after the fact. In addition, merges are difficult to track if code changes are merged as a new commit or through 'squashing' and 'rebasing' rather than through a traditional merge commit. As a consequence, despite best efforts, there will be inaccuracies in our classification of hard forks and individual commits, which we expect will lead to some underreporting of hard forks and to some underreporting of merged code.

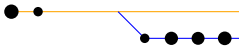
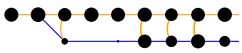
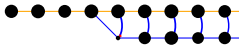
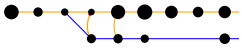
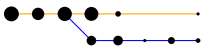
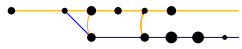
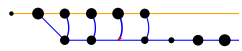
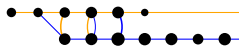
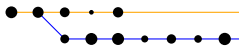
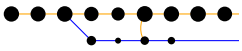
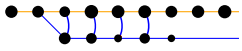
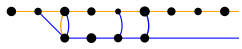
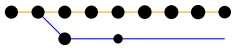
We analyze data with right-censored time series data, in which we can detect that projects have seized activity in the past, but cannot predict the future, thus seeing a larger chance for older forks to be discontinued.

Our study is limited to hard forks of which both fork and upstream repository are hosted on GitHub and of which the forking relationship is tracked by GitHub. While GitHub is by far the most dominant hosting service for open source, our study does not cover forks created of (typically older) projects hosted elsewhere and forks created by manually cloning or copying source code to a new repository. In addition, our interviews, as typical for all interview studies in our field, is biased toward answer from developers who chose to make their email public and chose to answer to our interview request, which underrepresented maintainers of upstream repositories in our sample.

4.3 Results

We explore practices and perceptions around hard forks along four facets that emerged from our interviews and data.

Table 4.2: Evolution patterns of hard forks

Id		Category	Total	Sub-category	Example	Count	Interviewees
1	Revive Dead Project	Success (F. active > 2 Qt.)	632	Upstream remains inactive		576	P12
2				Upstream active again		56	
3		Not success (F active <= 2 Qt)	420			420	
4	Forking Active Project	Both Alive	723	only merge		26	P10
5				only sync		107	P2, P13, P15
6				merge & sync		28	P9
7				no interation		562	P1, P3, P4, P5, P7, P14
8		Fork Lived Longer	7280	only merge		174	
9				only sync		686	
10				merge & sync		107	
11				no interaction		6313	P6, P8, P11
12		Fork does not out live upstream	6251	only merge		388	
13				only sync		762	
14				merge & sync		199	
15				no interaction		4902	

4.3.1 Frequency of Hard Forks

Our classifier identified 15,306 hard forks, confirming that hard forks are generally a rare phenomenon. As of June 2019, GITHUB tracks 47 million repositories that are marked as forks over 5 million distinct upstream repositories among GITHUB's over 125 million repositories.

Among those, the vast majority of forks has no activity after the forking point and no stars. Most active forks have only very limited activity indicative of social forks. Only 0.2 % of GITHUB's 47 million forks have 3 or more stars.

As our analysis of evolution patterns (Table 4.2) reveals, cases where both the upstream repository and the hard fork remain active for extended periods of time are not common (patterns 1, 2, and 4–7; 1157 hard forks, 8.8 %). Most hard forks actually survive the upstream project, if the upstream project was active when the fork was created (patterns 8–11; 7280 hard forks, 47.6 %), but many also run out of steam eventually (patterns 3 and 12–15; 6671 hard forks, 43.6 %).

While most hard forks are created as forks of active projects (patterns 4–15; 14254 hard forks, 93 %), there are a substantial number of cases where hard fork are created to revive a dead project (pattern 1–3; 1052 hard forks, 6.8 %), in some cases even triggering or coinciding with a revival of the upstream project (pattern 2; 56 hard forks, 0.36 %), but also here not all hard fork sustain activity (pattern 3; 420 hard forks, 2.7 %).

Discussion and implications

Even though the percentage of hard forks is low, the total number of attempted and sustained hard forks is not. Considering the significant cost a hard fork can put on a community through fragmentation, but also the potential power a community has through hard forks, we argue that hard forks are an important phenomenon to study even when they are comparably rare.

Whereas previous work typically looked at only a small number of hard forks, and research on tooling around hard-fork issues typically focus on few well known projects, such as the variants of *BSD* [180] or *Marlin* [138] or artificial or academic variants [84, 122], we have detected a significant number of hard forks, many of them recent, using many different languages, that are a rich pool for future research. We release the dataset of all hard forks with corresponding visualizations as dataset with this paper [15].

4.3.2 Why Hard Forks Are Created (And How to Avoid Them)

At a first glance, the interviewees give reasons for creating hard forks that align well with prior findings, including especially continuing discontinued projects or projects with unresponsive maintainers (P1, P2, P8), disagreements around project governance (P2, P12), and diverging technical goals or target populations (P3, P5, P6, P11, P13, P14, P17). As discussed, we identified 1052 hard forks (Table 4.2, patterns 1–3, 6.8 %) that forked an inactive project.

An interesting common theme that emerged in our interviews though was that many hard forks were not deliberately created as hard forks initially. More than half of our interviewees described that they initially created a fork with the intention of contributing to the upstream repository (social fork), but when they faced obstacles they decided to continue on their own.

Common obstacles were unresponsive maintainers (P1, P2, P8) and rejected pull requests (P11, P13, P14), typically because the change was considered beyond the scope of the project. For example, P2 described that "*before forking, we started by opening issues and pull requests, but there was a lack of response from their part. [We] got some news only 2 months after, when our fork was getting some interest from others.*" Similarly, some maintainers reported that a fork initially created for minor personal changes evolved into a hard fork as changes became more elaborate and others found them useful (P2, P14, P17); for example, P14 described that the upstream project had been constantly evolving and the code base became quickly incompatible with some libraries, so he decided to fix this issue while also adding functionality, after which more and more people found his fork and started to migrate.

Several maintainers also had explicit thoughts about how to avoid hard forks (both maintainers of projects that have been forked and fork owners who themselves may be forked), and they largely mirror common reasons for forking, i.e., transparent governance, being responsive, and being inclusive to feature requests. For example, P2 suggests that their project is reactive to the community, thus he considers it unlikely to be forked; similarly P16 decided to generally "*respond to issues in a timely manner and make a good faith effort to incorporate PRs and possibly fix issues and add features as the needs arrives*" to reduce the need for hard forks. Beyond these, P2 also mentioned that they created a contributing guide and issue templates to coordinate with contributors more efficiently; P14 suggested to "*credit the contributors*" explicitly in release notes in order to keep contributors stay in the community.

Discussion and Implications

Whereas forking was typically seen as a deliberate decision in pre-GITHUB days that required explicit steps to set up a repository for the fork and find a new name, nowadays many hard forks seem to happen without much initial deliberation. Social coding environments actively encourage forking as a contribution mechanism, which significantly lowers the bar to create a fork in the first place without having to think about a new name or potential consequences like fragmenting communities. Once the fork exists (initially created as social fork), there seems to be often a gradual development until developers explicitly consider their fork a separate development line. In fact, many hard forks seem to be triggered by rather small initial changes. These interview results align with the observation that only about 36 % of the detected hard forks on GITHUB have changed the project's name (cf. Figure 4.2a and 4.2b).[4]

More importantly, a theme emerged throughout our interviews that hard forks are not likely to be avoidable in general, because of a project's *tension between being specific and begin general*. On the one hand, projects that are more inclusive to all community contributions risk becoming so large and broad that they become expensive to maintain (e.g., as P17 suggests, the project maintainers need to take over maintenance of third-party contributions for niche use cases) and difficult to use (e.g., lots of configuration options and too much complexity). On the other hand, projects staying close to their original vision and keeping a narrow scope may remain more focused with a smaller and easier to maintain code base, but they risk alienating users who do

[4] An intervieweed hard-fork owners explained that they did *not* change the fork's name as a way to give credits to the upstream project, so not all hard forks without name changes should be automatically interpreted as being created through a gradual transition from social forks.

not fit that original vision, who then may create hard forks. One could argue that hard forks are a good test bed for contributions that diverge from the original project despite their costs on the community: If fork dies it might suggest a lack of support and that it may have been a good decision not to integrate those contributions in the main project.

In this context, a family of related projects that serve slightly different needs or target populations but still coordinate may be a way to overcome this specificity-generality dilemma in supporting multiple projects that each are specific to a mission, but together target a significant number of uses cases. However, current technology does not support coordination across multiple hard forks well, as we discuss next.

4.3.3 Interactions between Fork and Upstream Repository

Many interviewees indicate that they are interested in coordinating across repositories, either for merging some or all changes back upstream eventually or to monitor activity in the upstream repository to incorporate select or all changes. Some hard fork owners did not see themselves competing with the upstream project, but rather being part of a larger project. For instance, although fork owner P13 has over 1500 commits ahead of the upstream project, he still said that "*I would not consider it independent because I am relying on what they (upstream) are doing. I could make it independent and stop getting their improvements, but it's to their credit they make it very easy for their many hundreds of developers to contribute patches and accept patches from each other. They regulate what goes into their project very well, and that makes [merging changes] into my fork much easier.*" Some (P4 and P11) indicate that they would like to merge, once the reason for the hard fork disappears (typically governance practices or personal disputes). Also upstream maintainers tend to be usually interested in what happens in their forks; for example, P17, a maintainer of a project with thousands of (mostly social) forks, said "*I try to be aware of the important forks and try to get to know the person who did the fork. I will follow their activities to some extent.*"

However, even though many interviewees expressed intentions, we see little evidence of actual synchronization or merging across forks in the repositories: For example, P1, P4, P8, and P11 mention that they are interested in eventually merging back with the upstream repository, but they have not done so yet and do not have any concrete plans at this point. Similarly, P2, P6, and P10 indicate that they are interested in changes in upstream projects, but do not actually monitor them and have not synchronized in a long time. Our evolution patterns similarly show that synchronization (from upstream to fork) and merging (from fork to upstream) are rare. Only 16.18 % of all hard forks with active upstream repositories ever synchronize or merge (Table 4.2, patterns 4–6, 8–10, and 12–14).

What might explain this difference between intentions and observed actions is that synchronization and merging becomes difficult once two repositories diverge substantially and that monitoring repositories can becoming overwhelming with current tools. For example, P2 reports to only occasionally synchronize minor improvements, because the fork has diverged to much to synchronize larger changes; P10 has experienced problems of synchronizing too frequently and thus being faced with incomplete implementations and now only selectively synchronizes features of interest. In line with prior observations on monitoring change feeds [38, 61, 168, 249], interviewees report that systematically monitoring changes from other repositories is onerous

and that current tools like GITHUB's network graph are difficult to use and does not scale (P11, P16).

Discussion and Implications

Tooling has changed significantly since the pre-GITHUB days of prior studies on hard forks which may allow new forms of collaboration across forks: Git specifically supports merges across distributed version histories, as well as selectively integrating changes through a 'cherry picking' feature. GITHUB and similar social coding pages track forks, allowing developers to subscribe to changes in select repositories, and generally make changes in forks transparent [61, 62, 249]. Essentially all interviewees were familiar with GITHUB's network view [4] that visually shows contributions over time across forks and branches.

Even though advances in tooling provide new opportunities for coordination across multiple forks and project maintainers are interested in coordinating and considering multiple forked projects as part of a larger community, current tools do not support this use case well. Current tools work well for short-term social forks but tend to work less well for coordinating changes across repositories that have diverged more significantly.

This provides opportunities for researchers to explore tooling concepts that can monitor, manage, and integrate changes across a family of hard forks. Recent academic tools for improved monitoring [168, 249] or cross-fork change migration [180, 183] are potentially promising but are not yet accessible easily to practitioners. Also more experimental ideas about virtual product-line platforms that unify development of multiple variants of a project [17, 84, 157, 192, 211] may provide inspiration for maintaining and coordinating hard forks, though they typically do not currently support the distributed nature of development with competing hard forks. A technical solution could solve the specificity-generality dilemma (cf. Sec. 4.3.2), allowing subcommunities to handle more specific features without overloading the upstream project and without fragmenting the overall community. We believe that our dataset of 15,306 hard forks can be useful to develop and evaluate such tools in a realistic setting.

4.3.4 Perceptions of Hard Forking

Our discussion with maintainers confirmed that the line between hard forks and social forks is somewhat subjective, but, when prompted, they could draw distinctions that largely mirror our definition (long-term focus, extensive changes, fork with own community). For example, P2 agree that his fork is independent from the upstream project because they have different goals, and suggests the fork has better code quality, and better community management practices; the only remaining connection are upstream bug fixes that he incorporates from time to time. Also, P6 considers his fork as independent, given a quicker release cycle and significant refactoring of the code base.

For most interviewees, the dominant meaning of a fork is that of a social fork. When asked about perceptions of forks, most interviewees initially thought of social forks and have strong positive associations, e.g., others contributing to a project, onboarding newcomers and finding collaborators, and generally fostering innovation. For instance, P6 described the advantages of social forking as "*it encourages developers to go in a direction that the original project may not*

have gone," and similarly P9 thought that "*it could boost the creative ideas of the communities.*" One interviewee also mentioned that for young projects primarily focused on growth, being forked is a positive signal, meaning that the project is useful to other people. Social forks were so dominant in the interviewees' mind as a default, that we had to frequently refocus the interview on hard forks. When asked specifically about hard forks, several interviewees raised concerns about potential community fragmentation (P4, P6, P17), worried about incompatibilities and especially confusing end users (P3, P9, P14, P17), and would have preferred to see hard-fork owners to contribute to the upstream project instead (P3, P8, P12). However, concerns were mostly phrased as hypotheticals and contrasted with positive aspects.

Many interviewed owners of hard forks do not see themselves competing with the upstream repository, as they consider that they address a different problem or target a different user population. For example, P10 described his fork as a *"light version"* of the upstream project targeting a different group of users.

While it is understandable that hard-fork owners see their forks as justified, also some interviewed owners of upstream projects had positive opinions about such forks. For example, P17 expressed that forks are good if there is a reason (such as a focus on a different target population, in this case beginners), and that those forks may benefit the larger community by bringing in more users to the project; P18 suggested even that he would support and contribute forks of his own project by occasionally contributing to them as long as it will benefit the larger community.

Discussion and Implications

Overall, we see that the perception of forking has significantly changed compared to perceptions reported in earlier work. Forking used to have a rather negative connotation in pre-GITHUB days and was largely regarded as a last resort to be avoided to not fragment the community and confuse users. With GITHUB's rebranding of the word *forking*, the stigma around hard forking seems to have mostly disappeared; the word has mostly positive connotations for developers, associated positively with external contributors and community. While there is still some concern about community fragmentation, it is rarely a concrete concern if there are actual reasons behind a hard fork. Transparent tooling seems to help with acceptance and with considering multiple hard forks as part of a larger community that can mutually benefit from each other.

We expect that a more favorable view, combined with lower technical barriers (Sec. 4.3.2) and higher expectations of coordination (Sec. 4.3.3) makes hard forks a phenomenon we should expect to see more of. However, positive expectations can turn into frustration (and disengagement of valuable contributors to sustain open source) if fragmentation leads to competition, confusion, and coordination breakdowns due to insufficient tooling.

With the right tooling for coordination and merging, we think hard forks can be a powerful tool for exploring new and larger ideas or testing whether there is sufficient support for features and ports for niche requirements or new target audiences (e.g., solving the specificity-generality dilemma discussed in Sec. 4.3.2 with a deliberate process). To that end though, it is necessary to explicitly understand (some) hard forks as part of a larger community around a project and possibly even explicitly encourage hard forks for specific explorations beyond the usual scope of social forks. We believe that there are many ways to support development with hard forks and to coordinate distributed developers beyond what social coding site offer at small scale today.

Examples include (1) an early warning system that alerts upstream maintainers of emerging hard forks (e.g., external bots), which maintainers could use to encourage collaboration over competition and fragmentation if desired, (2) a way to declare the intention behind a fork (e.g., explicit GITHUB support) and dashboard to show how multiple projects and important hard forks interrelate (e.g., pointing to hard forks that provide ports for specific operating systems), and (3) means to identify the essence of the novel contributions in forks (e.g., history slicing [135] or code summarization [249]).

4.4 Summary

With the rise of social coding and explicit support in distributed version control systems, forking of repositories has been explicitly promoted by sites like GITHUB and has become very popular. However, most of these modern forks are not hard forks in the traditional sense. In this Chapter, we revisited the question about the motivation for hard forks and explore whether they have changed with the rise of social coding. We believe it is necessary to revisit hard forking after the rise of social coding and GITHUB. Specifically, we aim to understand the hard-fork phenomenon in a current social-forking environment, and understand how perceptions and practices may have changed.

We automatically detected hard forks and their evolution patterns and interviewed open-source developers of forks and upstream repositories to study perceptions and practices. We found that perceptions and practices have indeed changed significantly: Among others, hard forks often evolve out of social forks rather than being planned deliberately and developers are less concerned about community fragmentation but frequently perceive hard forks a positive noncompetitive alternatives to the original projects.

This project is a complement to the previous project described in Chapter 3 to update and deepen our understanding regarding practices and perceptions around hard forks can inform the design of better tools and management strategies to facilitate efficient collaboration. With the right tooling for coordination and merging, we think hard forks can be a powerful tool for exploring new and larger ideas or testing whether there is sufficient support for features and ports for niche requirements or new target audiences. Moreover, it is necessary to explicitly understand (some) hard forks as part of a larger community around a project and possibly even explicitly encourage hard forks for specific explorations beyond the usual scope of social forks.

Chapter 5

New Intervention: Identifying Features in Forks (INFOX)

This chapter shares material with the ICSE'18 paper "Identifying Features in Forks" [249] and ICSE'18 - Poster paper "Forks Insight: Providing an Overview of GitHub Forks" [182].

In previous chapters, we observed differences between projects regarding the degrees of collaboration efficiency using fork-based development mechanisms, thus, we studied *natural interventions* that are correlated to higher collaboration efficiency and tested the feasibility of applying such intervention to a broader population (see Chapter 3). However, we also observed that some existing interventions (*e.g.*, GITHUB network view and GITLAB fork list view shown in Figure 5.1 for presenting an overview of the community) are not good enough. In this chapter, we describe our first *tooling intervention* to improve the awareness of a community and generate a better overview for fork-based development mechanism. We design an approach to identify unmerged cohesive code changes (named features) from forks. The approach is called INFOX, which is short for **I**de**N**tifying **F**eatures in f**Or**K**S**.

5.1 Motivation

As described in Section 1, because the number of forks of a project is large, it is hard to maintain an overview of the whole community, which would lead to other problems. Several open-source developers that we interviewed for this paper indicated that they are interested in what happens in other forks, but cannot effectively explore them with current technology, such as GitHub's network graph shown in Figure 5.1b: *"I care, but, it is very hard to track all of the forks."* This developer is using SourceTree, which visualizes commit history of a repository through GUI, to explore code changes in other forks one by one, and he said *"it is just difficult"* [P5]; *"I do not have much visibility of the forks. They are too many, and it is overwhelming to keep track of them"* [P9]. The difficulty to maintain an overview of forks leads to several additional problems, such as redundant development, lost contributions and suboptimal forking point (as described in Section 1).

GitHub's main facility to navigate forks is the network view (Figure 5.1b), which visualizes the history of commits over time across all branches and forks of a project. This cross-fork visualization provides transparency to developers who want to track ongoing changes by others, want to know who is active and what they are trying to do with the code [61]. For example, one of the developers we have interviewed said: *"I check the more updated forks. I think this view is helpful, because I am not gonna look at all 60 forks. 60 is a lot, probably this project has thousands, that will be ridiculous. I will never do that"* [P4].

Although the network view is a good starting point to understand how the project evolves, it is tedious and time consuming to use if a project has many forks. In order to see older history, users click and drag within the graph, and if users want to see the commit information, they hover the mouse over each commit dot and read the commit message. Also, they *"have to scroll back a lot to find the fork point and then go to the end again for seeing what changed since then in the parent and in the fork"* [7]. If developers want to investigate the code changes of certain forks, they have to manually open and check each fork. As one developer stated *"I don't look at the graphs on GitHub... it is very hard to find the data, you have to scroll for 5 minutes to find stuff"* [P5]. The view does not even load when there are over 1000 forks, no matter they are active or inactive.

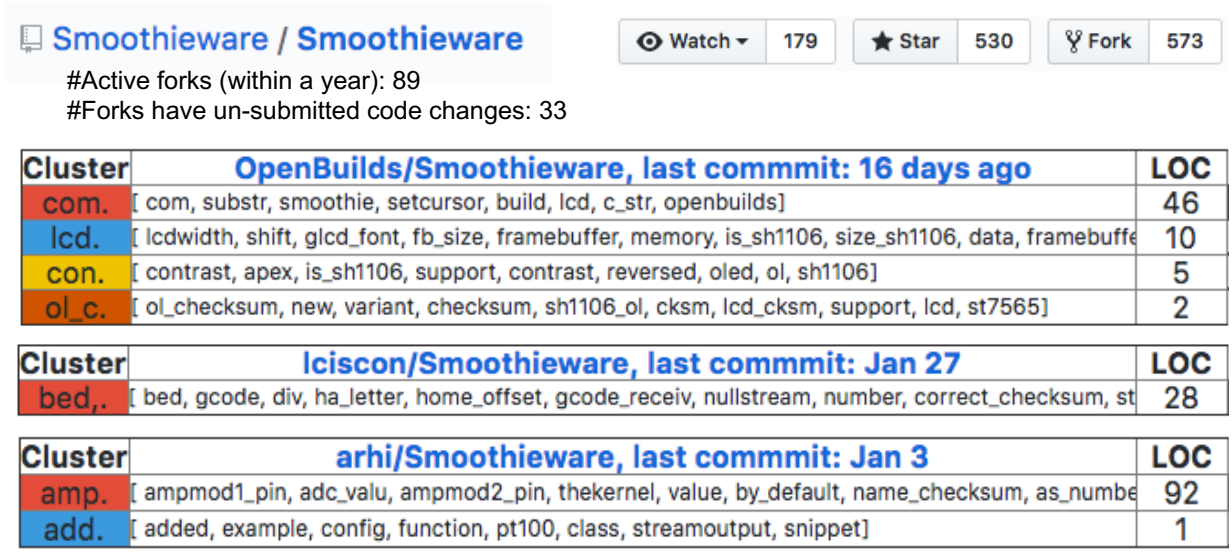
The goal of our work is to identify and label cohesive code changes, *features*, among changes in forks to provide a compact overview of features and their implementations. This is a step to establish an overview of development activities in various forks of a project.

In contrast to GitHub's network view (Figure 5.1b), we deemphasize commits, which frequently have unreliable descriptions and frequently are unreliable indicators of cohesive functionality, as it is common that commits tangle code of multiple features and even more common that a single feature is scattered across multiple commits [23, 112, 113, 127, 135, 159]. Instead, we *cluster changed code* based on relationships and dependencies within those code fragments and label each feature with *representative keywords* extracted from commit messages, code, and comments. Technically, we take inspiration from CLUSTERCHANGES [23] to untangle code changes during code review based on a graph of code dependencies and repurpose the idea for our problem; furthermore we incorporate community-detection techniques [95] to refine an initial clustering and information-retrieval techniques [197] for deriving concise labels (See Figure 5.1a).

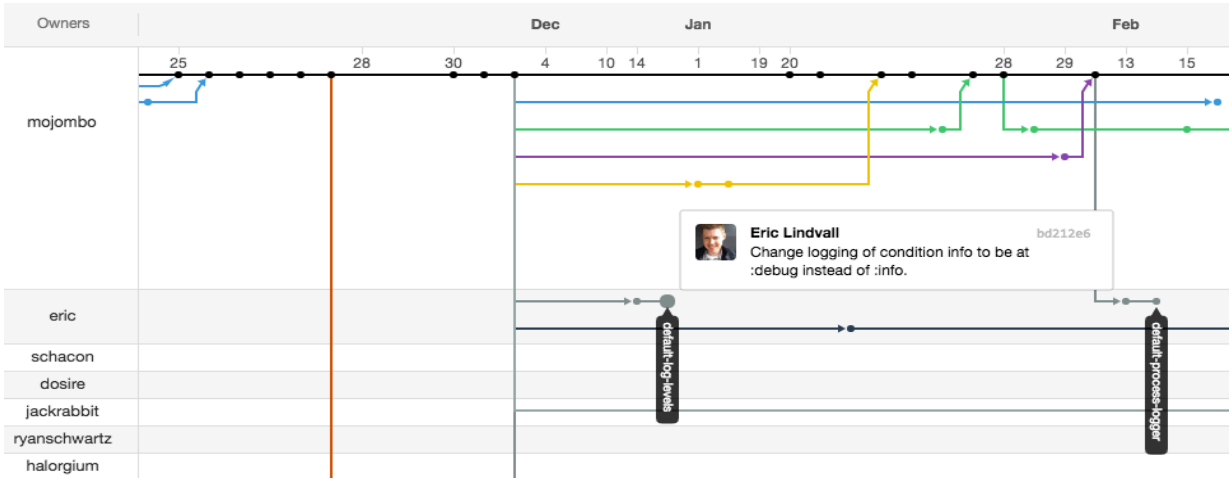
To summarize, we contribute (a) INFOX, an approach and corresponding tool, which automatically identifies and summarizes features in forks of a project, using source code analysis, community detection, and information-retrieval techniques, and (b) evidence that INFOX improves accuracy over existing techniques and provides meaningful insights to maintainers of forks.

5.2 Method

INFOX identifies and labels features within a larger change of a fork. It takes the code change difference between the latest commit of the upstream (source snapshot) and the latest commit of



(a) INFOX's overview summarizes features in active forks.



(b) GITHUB's network graph shows commits across known forks, but is difficult to use to gain an overview of activities in projects with many forks.

Figure 5.1: Complementary solutions for lack of overview problem in fork-based development.

the fork (target snapshot) from GitHub, which returns the non-merged changes from fork.[1] Then it proceeds in three steps (as shown in Figure 5.3):

- Identify a dependency graph among all added or changed lines of code by parsing and analyzing the code for multiple kinds of dependencies (Sec. 5.2.1).
- Cluster the lines of the change based on the dependency graph using a community-detection technique, mapping each line of code to a feature, such that lines with many connections in the dependency graph are mapped to the same feature (Sec. 5.2.2).
- Label each cluster by extracting representative keywords with an information-retrieval technique (Sec. 5.2.3).

The first step is inspired by CLUSTERCHANGES, an approach to untangle code in commits for code review [23]. CLUSTERCHANGES clusters changed code fragments based on a dependency graph of lines of code. We adopt this idea for a different purpose (identifying and naming features in multiple forks rather than untangling changes in a single commit) and we extend the approach with additional kinds of dependency edges, additional steps in the clustering process, and labeling of clusters, as we will explain.

5.2.1 Generating a dependency graph

We generate a dependency graph for all lines of code of the target snapshot by parsing the target snapshot and analyzing the resulting abstract syntax tree. We add edges between lines for several kinds of relationships of code fragments within those lines that may indicate that the two fragments are that are more likely to be related. We collect the following kinds of dependencies, which we also illustrate on a simple excerpt of an email system in Figure 5.2:

- *Definition-usage edges:* We add edges between the definition and use of functions and variables in the program, and the definition and use of structs or classes and their members. We conjecture that def-use relationships between two code fragments often point to two code fragments that fulfill a joint purpose and are thus more likely to be part of the same cohesive change in a larger change.
- *Control-flow edges:* We generate a control-flow graph for the source code and add edges between two lines if there is a control dependency relation between the statements of each line. In line with Emerson's cohesion metrics [79], we think that the flow of control information contributes to the cohesion of code changes.
- *Adjacency and hierarchy edges:* We add edges between consecutive lines and lines that represent hierarchical structures in the source code (struct/class members point to the outer struct definition). Adjacency edges and hierarchy edges represent the structure of the source code and indicate that code fragments that are located close to each other are more likely to belong to the same cohesive fragment than code fragments scattered across different places.

The result is a labeled, weighted, undirected graph, in which nodes represent lines of code and edges represent the identified dependencies listed above. We assign a low weight of 1 for adjacency edges, and a weight of 5 for all other edges. Intuitively, semantic dependencies in the

[1] While developed for changes in a single fork, our approach can be technically used to cluster the changes between any two code snapshots, including two commits in a single repository or two copies of code maintained without a version control system.

program should be stronger indicators of features than structural relations. We use an undirected dependency graph, as our experiments showed no benefit in maintaining directionality.

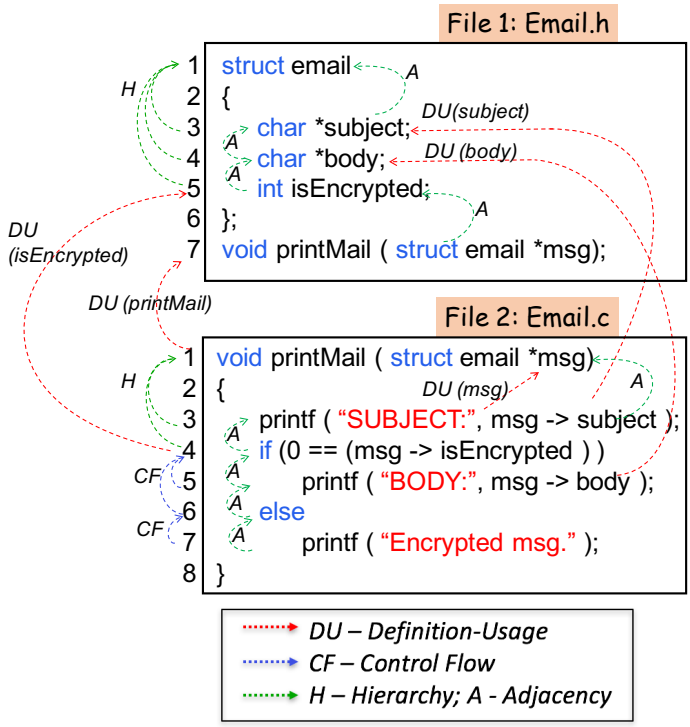


Figure 5.2: Edge examples of an email system.

Using a *diff* command between the source and the target snapshot, we identify and mark all nodes that have been added or changed in the target snapshot (highlighted in Figure 5.3).

Compared to CLUSTERCHANGES [23], we add edges between nodes with hierarchical and control-flow relations, and add weights.

5.2.2 Identifying features by clustering the graph

Given the labeled, weighted, undirected dependency graph of the target snapshot, we identify clusters of nodes that have many edges within the cluster but few edges across clusters (known as *community detection* in the network analysis community [86, 218]). We interpret each cluster and the corresponding lines of code as a feature. We start with an initial simple clustering step, but provide additional means to further split and join clusters that can be used in an interactive tool or applied automatically using heuristics.

Initial clusters. In line with CLUSTERCHANGES [23], we establish initial clusters by removing all unlabeled nodes (see Figure 5.3) from the dependency graph (i.e., all nodes that have not been added or changed between source and target) and by detecting *connected components* in the resulting graph. Each connected component is considered as a feature.

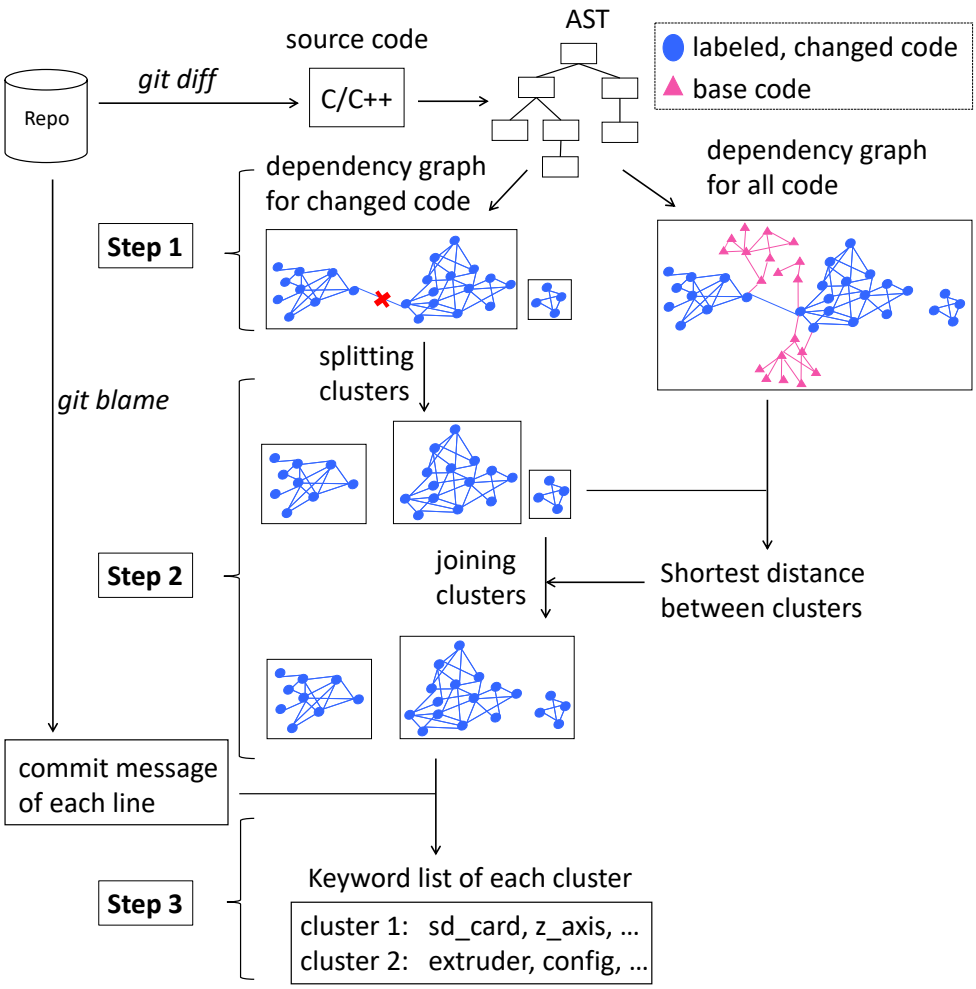


Figure 5.3: Generating and clustering dependency graphs to identify features, and labeling features [249].

This simple splitting works well for many changes, for example, grouping together adjacent code fragments and new function definitions with their corresponding calls. Unfortunately, for other changes, including large and tangled changes—that we see more frequently in forks than in individual commits—this initial clustering is susceptible to problems where two unrelated features are merged, just because their implementations share a single adjacency edge in one place in the source code. Similarly, some code fragments may belong together but have no link within the changed code, such as multiple scattered code fragments calling the same previously existing logging function. For that reason, we go beyond the work of CLUSTERCHANGES [23] and provide additional (optional) support for splitting and joining clusters further.

Splitting clusters. *Community detection* identifies modules in a graph according to the structural position of nodes [86]. In community detection, the key optimization criterion is to maintain more connections within the module than across modules. We use community detection to split

large clusters into smaller ones that are only loosely connected.

We adopt a state-of-the-art community-detection algorithm by Girvan and Newman [95]. Its idea is to count the number of shortest paths between node pairs. This count, weighted by the edge weight, is called the *edge-betweenness* score. The edges with higher betweenness tend to be the bridge between communities. The clustering algorithm iteratively removes the edges with the highest *edge-betweenness* score from the original graph.

In the example of Figure 5.3 (Step 1), the highlighted edge is the one with the highest betweenness score, bridging two otherwise highly interconnected clusters. Our algorithm removes this edge, splitting the large cluster into two smaller ones (Step 2).

Note that community detection has no natural stop criteria. The algorithm can continue until the last edge is removed, creating singleton clusters. In practice, several heuristic stop criteria exist, such as maximizing a modularity metric [95] or stopping when a given maximum number of cut edges do not yet result in a new cluster. Since, despite experiments, we could not identify a single robust stop criterion for our problem, we primarily envision splitting in an interactive setting, in which developers can request to split a large feature into two using the community-detection algorithm, if they judge this to be beneficial.

Joining clusters. Scattered implementations of a single conceptual feature may result in graph components without any connecting edge. In our experience, this sometimes generates sets of small clusters that appear to be highly related (e.g., call the same function, use the same variable), but have no dependency edge within the changed code. To identify these as a single feature nonetheless, we analyze how those clusters are related in the context of the entire implementation, not just the added or changed lines of code.

To this end, we compute the distance between two clusters in the entire dependency graph that includes the unlabled nodes representing unchanged lines of code that are the same in the source and target snapshot. Given two clusters, we compute the distance of two clusters as the shortest distance between any pair of nodes, in which each node belongs to one of each clusters. In our example, in Figure 5.3, the two initial clusters are separated by only a single unmarked node, indicating that they might be joined.

Again, there is flexibility in selecting thresholds about when to join two clusters. In addition to interactive mechanisms, we apply joins by default for pairs of clusters that are separated by a single unlabled node when at least one of the clusters is smaller than 50 nodes (lines of code). We arrived at this default threshold after observing, across a large number of forks, that small clusters are more frequently affected by this, whereas large clusters are more likely to already share an edge.

5.2.3 Labeling features

After identifying features with clusters in the dependency graph of the changed code (possibly with additional splitting and joining), we can already show the clusters to developers. However, to allow them to gain an overview of a fork's changes quickly without having to read a large amount of source code, we label each feature with representative keywords.

```
1     if (dual_x_carriage == DXC_DUPLICATION){
2         setTarget(duplicate_extruder_temp);
3         duplicate_extruder_temp_offset = code_value();
4         duplicate_extruder_x_offset = max(i, t);
5         SERIAL_ECHO(extruder_offset[0]);
6         extruder_duplication_enabled = false;
7     }
```

Figure 5.4: Source code excerpt from Marlin.

In contrast to GitHub's network graph (see Figure 5.1b), which only shows individual commit messages, we compute representative keywords from multiple sources. We use commit messages in the process, but do not rely on them, because (1) commit messages are often too verbose to consume quickly, because (2) as discussed, commits do not always align with features, and because (3) we do not consider the text of commit messages as reliable descriptors. Instead, we use information-retrieval techniques to identify keywords that are distinctive for a given feature, meaning that those keywords represent the feature better than other features or the base implementation. Specifically, we proceed in three steps:

- First, we collect a corpus of text for each feature and an additional corpus for the unmodified source code. For each feature, the corpus contains (verbatim) all lines of code associated with this feature, including variable names and function names. We also include source-code comments that may provide additional explanations. Comments are added to corresponding clusters based on their line number. Finally, we identify the commits that introduced the changed lines of a feature (using 'git blame') and add all corresponding commit messages to the corpus.
- Second, we tokenize each corpus (e.g., splitting variable names at underscores [44]) and perform the standard normalization techniques of stemming (e.g., unifying variations of words such as duplicate, duplicated, and duplication) and removing stop words (specifically reserved keywords such as *int, sizeof, switch, and struct*).
- Third, we identify keywords that are important in one corpus as compared to all other corpora using the well-known *Term Frequency Inverse Document Frequency (TF-IDF)* scoring technique [197]. The importance of a keyword (its TF-IDF score) increases proportionally to the number of times a word appears in the feature's corpus but is offset by the frequency of the word in the other feature's corpora [133]. We calculate the TF-IDF score of each word and of each 2-gram (unique sequence of two words [215]) in the feature's corpus. We report the five highest scoring words and five highest scoring 2-grams as labels for the feature.

For example, consider the code snippet from the Marlin 3D printer firmware[2] in Figure 5.4, which we represent by the relevant keywords *duplicate_extruder, extruder_temp, offset, dual_x, x_carriage*, which are common in this code fragment but not elsewhere in the firmware implementation.

We arrived at our solution of tokenizing composed variable names (with underscore) and

[2] https://github.com/MarlinFirmware/Marlin

Figure 5.5: Features in fork *DomAmato/ofxVideoRecorder*; tree view displaying hierarchical relation between split features; colors related code to features.

using 2-grams after some experimentation. On the one hand, composed variable names (e.g., *duplicate_extruder_x_offset* in Figure 5.4) are often too specific and dominate the TF-IDF score, such that all keywords are long and often similar variable names. On the other hand, we do not want to discard them entirely as they often include descriptive parts that represent the feature. Finally, tokenizing all composed words sometimes leads to overly generic words, for example, unable to distinguish the different kinds of extruders in 3D printers. Tokenization combined with 2-grams provides a compromise that can pick up common combinations of words without relying too much on specific long combinations and generic short words.

5.3 Implementation & User Interface

We implemented INFOX for C and C++ code in a tool of the same name. INFOX takes a link to a GitHub project and collects all active forks. For each fork, it downloads the latest revision of each as target snapshot. Unless instructed otherwise, it takes the latest revision of upstream repository as that fork's source snapshot. As output, INFOX produces an HTML file that contains

summaries of features and keywords for all analyzed forks, ranked by the time of their last commits, as shown in Figure 5.1a. In addition, for each fork, it produces an HTML file that maps the features to source code (using colors, similar to FeatureCommander [83]) as shown in Figure 5.5. Navigation buttons allow to jump between scattered code fragments of a feature.

To build a dependency graph, INFOX parses C/C++ code with srcML [54] and performs a lightweight name-resolution analysis to detect def-use edges. Since reliably identifying all such edges in a complex language as C++ is difficult, our implementation is unsound, but provides a fast and sufficient approximation for our experiments. For example, INFOX does not disambiguate function pointers or other advanced language constructs.

Splitting and joining is currently implemented such that developers can interact with the web page and select which additional features to split and which to join. Splits are currently precomputed for features larger than 50 lines, as are joins for small features (by generating multiple static HTML pages through which the user navigates). This can easily be replaced by on-demand computations on a web server. Split clusters are illustrated with a hierarchy allowing users to track and undo splits. Our source code is publicly available at `https://github.com/shuiblue/INFOX`.

5.4 Evaluation

We evaluate INFOX with regard to effectiveness and usability. Specifically, we address four research questions:

- **RQ1: To what extent do identified clusters correspond to features?** We measure the effectiveness of INFOX's clustering approach by comparing how well the clusters match previously labeled features in the code. To that end, we will establish a ground truth of features in multiple code bases. We further compare the result of INFOX with the state-of-the-art (called CLUSTERCHANGES) [23].
- **RQ2: What design decisions in INFOX are significant to cluster cohesive code changes?** We aim to understand the factors that influence the effectiveness of INFOX. Specifically, we investigate how sensitive INFOX is to different kinds of edges in the dependency graph and to the splitting and joining steps.
- **RQ3: To what extent do developers agree with INFOX's clustering result?** Complementing RQ1, we explore whether fork maintainers in open-source projects agree with how INFOX divides and labels their own contributions.
- **RQ4: Can INFOX help developers to gain a better overview of repository forks?** We investigate whether INFOX helps developers to gain new and useful information about a project's many forks, such as recognizing useful contributions or redundant development in other forks.

We answer the first two research questions in a *controlled setting*, in which we *quantitatively* measure the accuracy of different clustering strategies on a number of subject systems for which we establish some ground truth as benchmark. Subsequently, we *qualitatively* answer the remaining two research questions in a *human-subject study*, in which we discuss INFOX's results with 11 developers of forks of popular open source systems. The studies are complementary, allowing us to both (a) systematically explore a large number of diverse scenarios while control-

ling several confounds and deliberately exploring the effects of changing independent variables (internal validity), as well as (b) validate in a practical setting how developers can benefit from the approach in their day-to-day development (external validity).

5.4.1 Quantitative Study (RQ1 & RQ2)

In a first study, we answer RQ1 and RQ2 in a controlled setting by quantitatively comparing clustering results of INFOX and CLUSTERCHANGES against a ground truth of known features in a number of open-source projects.

Establishing ground truth. A key challenge in evaluating approaches that identify features and cohesive code fragments (including a vast amount of literature on the concept-location problem) is to establish ground truth—a reliable data set defining which code fragments belong to which features. Once such ground truth is established, it is easy to define an accuracy measure and to compare different approaches and their variants. There are many different ways to establish ground truth, each with their own advantages and disadvantages, including (1) asking researchers or practitioners to manually assign features to code fragments [74, 189] (possibly biased and subjective, possibly low inter-rater reliability, expensive), (2) using indirect indicators such as code committed in a single commit or by a single author [68, 242] (questionable reliability), (3) using results of other tools as reference [115, 116, 186] (questionable reliability), or (4) using existing traceability mappings created for compliance reasons [52, 53, 58, 59, 60, 68] (uncommon practice outside industrial safety-critical systems). In this paper, we use a new, different approach and use existing mappings of code fragments to features through #ifdef directives in C/C++ code.

The preprocessor is commonly used in C/C++ code to implement optional features and support portability such that users can customize their builds by instructing the preprocessors which features to include [80, 137, 148]. We argue that *#ifdef*-guarded code fragments are often good approximations of features (extensions, alternatives) that developers might add in a fork of a software system. In fact, in some systems like the Marlin 3D-printer firmware, developers often add *#ifdef* guards around code blocks that they integrate back into the upstream repository as pull requests [214].

Given a C/C++ project, we identify all preprocessor macros that correspond to features of the system, excluding macros that are used for low-level portability issues. For each macro, we identify all code fragments that are guarded by this macro. We consider this macro-to-code mapping as the ground truth for features in the experiments, as illustrated in Figure 5.6. Extracting ground truth from preprocessor annotations has the advantage that those annotations have been added by practitioners independently of our experiments and that they can be extracted automatically at scale. As developers typically want to compile the code with and without those features, the mapping is typically well maintained as part of normal development activities and the features correspond to units of implementations that developers and users care about. Note that preprocessor annotations do not map all of the project's code to features, but this is not necessary, because we only want to cluster the code changed in forks. We describe next how we simulate such changes to forks.

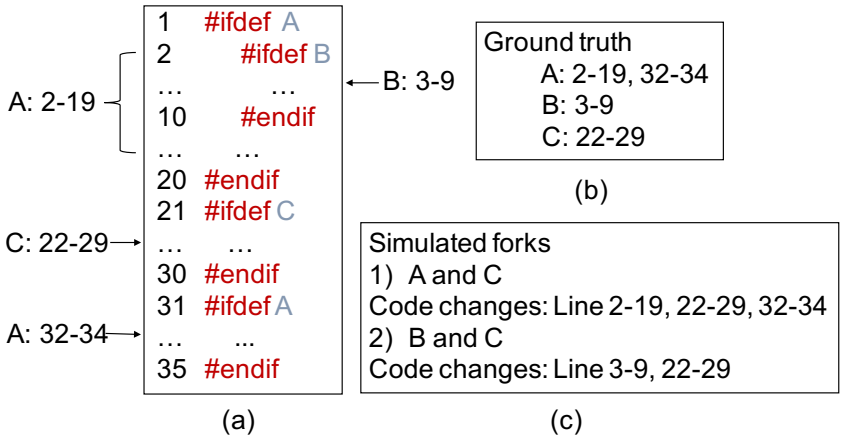


Figure 5.6: Extracting preprocessor-based ground truth and simulating forks.

Simulating forks. For a given project, we simulate multiple forks, of which each adds multiple features. To that end, we select a subset of features in the project and create the *source snapshot* by removing all code corresponding to those features (based on the ground-truth mapping), whereas we remove only the #ifdef directives but not the corresponding implementations from the *target snapshot*. That is, source and target snapshot differ exactly in the implementation of the selected features. We then evaluate whether INFOX can cluster the changed code into the features originally defined by the project's developers. By selecting different sets of features, we can generate different simulated forks for the same project.

Since INFOX divides all code added in a fork into non-overlapping clusters, we avoided nested macros in one feature combination when generating simulated forks. Technically, we select macros incrementally and randomly, and discard any macro for which code overlaps with previously selected macros until we found the desired number of non-overlapping macros. For example, in Figure 5.6, macro B is nested in macro A, thus we may generate simulated forks with A and C, and B and C, but not forks with A and B.

In order to evaluate the *effectiveness* and *robustness* of INFOX, we ran INFOX on multiple projects and on multiple simulated forks per project. We explored all combinations of the following three *experimental parameters* and generated 10 simulated forks for each combination, resulting in 156 simulated forks per project.

- *Number of macros:* We selected between 3 and 15 macros per simulated fork, simulating smaller and larger changes.
- *Proximity:* We either selected all macros (a) from the same file or (b) from different files, simulating more and less heavily tangled features.
- *Feature size:* We sorted all features by size (lines of code) and split them equally into smaller and larger features. We then sampled either (a) twice as many large features than small features or (b) twice as many small features than large features, thus further varying simulated forks.

Table 5.1: Subject projects

Software System	Domain	LOC	#F	F-LOC
Cherokee	web server	51,878	328	7,679
clamav	anti-virus program	75,345	285	10,809
ghostscript	postscript interpreter	442,021	816	21,864
Marlin	3D printer firmware	190,799	280	26,395
MPSolve	mathematical moftware	10,181	17	1524
openvpn	security application	38,285	276	23,288
subversion	revision control system	509,337	409	28,443
tcl	program interpreter	135,183	2,481	26,618
xorg-server	X server	527,871	1,360	95,227
xterm	terminal emulator	49,621	453	19,208

LOC: lines of code; #F: number of unique features macros;
F-LOC: size of features in LOC

Subject systems. We use open-source software systems implemented in C/C++ with #ifdef annotations. We selected projects differing in domain, size, and number of features from existing research corpora [137, 214]. Table 5.1 lists the 10 selected systems.

Accuracy (dependent variable). To evaluate how well a clustering result matches the ground truth, we use a standard accuracy metric from community detection [218]: Considering all possible pairs of nodes ($2n(n-1)$ pairs for n nodes), accuracy is the ratio of correctly clustered pairs (denoted as CCPs) among all the pairs of nodes ($accuracy = \frac{CCPs}{2n(n-1)}$). A pair is correctly clustered if two nodes that belong to one community in the ground truth are assigned to the same community in the result, and if two nodes from different communities are assigned to different communities. Let Boolean function $G(i, j)$ denote whether, in the ground truth, node i and node j are in the same community, and $C(i, j)$ denote whether, in the clustering result, node i and node j are in the same cluster. A pair is correctly clustered iff $G(i, j) = C(i, j)$. Note that this measure does not require a direct correspondence of clusters, but measures to what degree pairs of lines of changed code in a simulated fork are correctly assigned to the same or differing features.

Independent variables. For RQ1 and RQ2, we compare the accuracy of INFOX when changing which subsets of edges to consider and whether to perform splitting and joining. As automated stop criteria for splitting, we stop after 5 additional clusters; for joining, we use our default stop criterion described in Section 5.2.2. Furthermore, we consider how CLUSTERCHANGES would perform if used for this problem unmodified (conceptually equivalent to INFOX without further splitting and joining and limited to consecutive lines and def-use dependencies in the clustering process). We used the Paired Wilcoxon rank-sum test to establish statistic significance.

Threats to validity. External validity is bound by the use of simulated forks, that provide ground truth for realistic settings but are not real forks. The elimination of nested macros

may make simulated forks to be cleaner than real forks. Nonetheless, we select systems from different domains with different number of macros which are heavily based on industry-strength technologies. Besides, we did not rely on the ifdef evaluation alone, but triangulated our results with the user study (see Sec. 5.4.2). INFOX is conceptually entirely independent of the programming language. With respect to implementation, generalizations to other languages than C/C++ should be done with care.

Regarding internal validity, our reimplementation of CLUSTERCHANGES may not be faithful, but was unavoidable as the original tool is not publicly available. To keep the design space manageable we do not explore different weights and stop criteria, but have only done initial sensitivity analyses to establish that the results are robust with regards to other weights or minor changes in stop criteria.

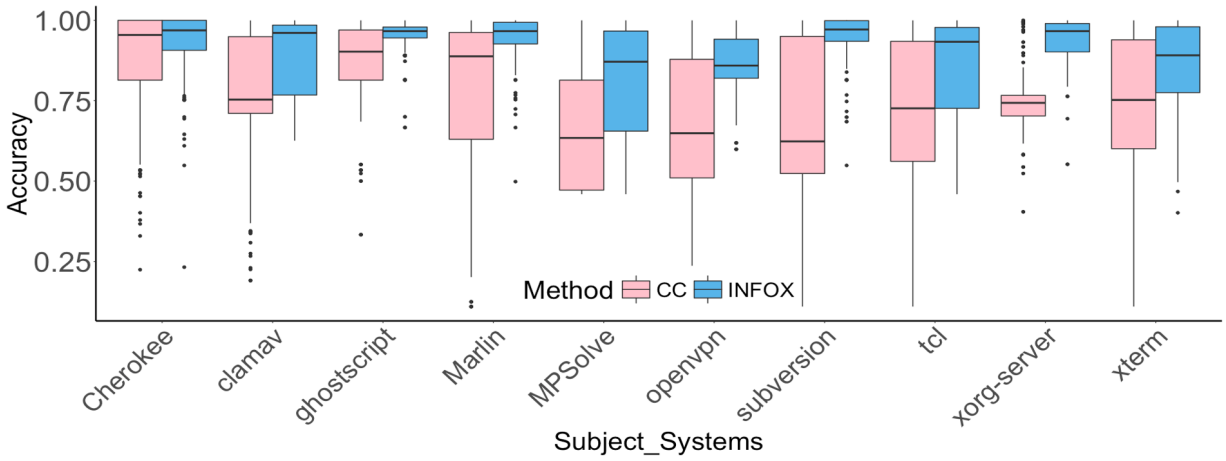


Figure 5.7: Accuracy of INFOX and CLUSTERCHANGES (CC) for 10 projects, 156 simulated forks units each.

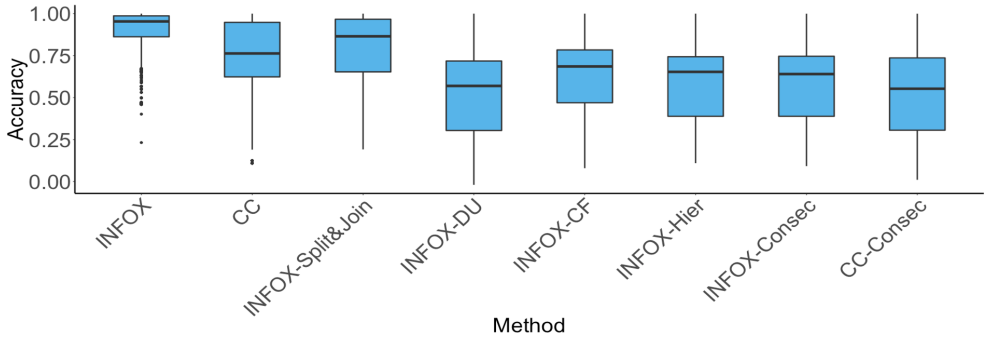


Figure 5.8: Accuracy across all 1560 simulated forks for different variations.

Results. We show the accuracy results in Figure 5.7 aggregated over 1560 simulated forks of all 10 subjects. **Regarding RQ1, we conclude that INFOX assigned features with 90 % accuracy**

and improves accuracy over CLUSTERCHANGES by 54-92%. The results are stable across all 10 projects and statistically significant ($p < .05$). In 102 simulated forks (6.5%), INFOX achieves a much higher accuracy than CLUSTERCHANGES (e.g,. accuracy increased 0.5), of which 61 cases are due to splitting and 41 cases are due to joining.

In Figure 5.8, we show accuracy results of all 1560 simulated forks split by different variations, specifically different kinds of dependency edges and with and without splitting and joining. **Regarding RQ2, we observe that splitting & joining steps improves accuracy by 4-14 %** (stat. sign., $p < .05$). Removing any kind of edges from the clustering approach significantly affects accuracy as well ($p < .05$); **all kinds of edges are important for the clustering quality**, but the definition-usage edges are the most influential ones.

5.4.2 Human-subject study (RQ3 & RQ4)

To evaluate the usability of INFOX, we contacted open-source developers who maintain forks to validate identified features and explore whether the generated summaries provide meaningful insights.

Study design. We invited developers of active forks (see selection below) for a remote interview. We conducted each interview in a semi-structured fashion divided into four phases:

- *Opening and introduction:* We started each interview by briefly explaining our research topic and the general purpose of our study. We asked whether the participants would share their screen with us and whether they consent to screen and audio recording.
- *Validating clustering result (RQ3):* In order to help participants remember what the code changes are, and also help us to gain domain knowledge for a better conversation, we first asked participants to briefly describe the project and code changes. Subsequently, we sent them the clustering result of INFOX for their own fork as a folder of HTML files (as illustrated in Figure 5.5). Within those results, participants could split and join clusters interactively. We started with an initial clustering result (without any splitting) and explained how to read and navigate the results.
  In a subsequent discussion, we pursued two questions: Whether the keywords are representative of their feature implementation and whether the clustering of the source code is meaningful to them. Most of the participants were communicative, and right after spending some time learning how to interact with INFOX, they started to navigate among code changes, explaining the meaning of the code, and whether clusters made sense or not. In line with methods for think-aloud protocols [118], we encouraged participants that were interacting with INFOX without saying anything for a long time to speak out loud, asking probing questions, such as *"Could you tell us what are you looking at?"* or *"Would you explain what this code cluster means?"*
- *Exploring the project overview (RQ4):* Before exploring INFOX's summary of other forks, we transitioned the discussion with the question *"Do you check what other forks are doing in this project?"* and followed up with questions on how and for what purpose they do this. Afterward, we sent them the project overview (cf. Figure 5.1a) and encouraged them to look through the list of forks. By clicking on the name of a fork, they could also

Table 5.2: Participants of our user study and their projects

Project	#Forks	#Active Forks	LOC	Change size (LOC)	Domain	Participant
MarlinFirmware/Marlin	4149	1901	19,799	2-3753	3D printer	P1 P3
Smoothieware/Smoothieware	566	237	61,425	19-11, 263	3D printer	P5 P6 P7
grpc/grpc	2226	470	95,838	3-480,901	general-purpose RPC framework	P2
timscaffidi/ofxVideoRecorder	60	24	611	7-23,228	video recording extension	P4
arduino/Arduino	5592	669	112,692	23-7,643	electronic prototyping platform	P4
bitcoin/bitcoin	9696	1242	99,746	6-647	experimental digital currency	P8 P9 P10
ariya/phantomjs	4,921	749	10,031	45-2,358	Scriptable Headless WebKit	P11

explore that fork's code with INFOX's results, just as they previously did for their own fork. Participants were usually actively exploring other forks at this point without our prompting and shared discoveries with us. When participants explored the code of a fork, we asked whether the keyword summary provided them a reasonable approximation of what they found in the implementation. In addition, we opportunistically asked questions about the relevance of keywords and the accuracy of clustering results in other forks based on their understanding (similar to questions about their own fork previously) when it fit the flow of the exploration.

- *Open discussion and closing:* We concluded each session with general and open-ended questions about further use cases and suggestions for improvement.

We compensated each participants with a $10 Amazon gift card. The interviews lasted between 30 and 90 minutes.

Participant selection. We searched for projects with active forks using two strategies. First, we used the GitHub search to find projects written in C/C++, selecting projects with more than 30 forks. Second, we queried GHTorrent [98] for the 100 C/C++ projects with the most first-level forks.

Among these projects, we selected forks that: (a) had at least one commit within the last year (increasing the chance that interviewees can remember their changes), (b) have added at least 10 lines of code (smaller changes are less likely to be a feature implementation), (c) have a large portion of commits submitted by the fork owner (excluding forks that aggregate changes of others), and (d) have a public email address or website of the fork owner. To enable questions about the overview page, we excluded projects for which we could not find at least three forks that fit these criteria.

In the end, we analyzed 58 projects on Github and found 12 projects fit our filtering criteria. We identified 81 fork owners. We sent out an email to candidate developers briefly describing our study. We interviewed 11 developers from 7 different projects (response rate 13.6%). We quickly reached saturation in that additional interviews provided only marginal additional insights. In Table 5.2 we list the characteristics of the projects from which we interviewed developers. All developers are experienced open-source developers.

Analysis. We analyzed the interviews primarily qualitatively, analyzing what participants learned and how they interacted with the tool. Two of the authors transcribed and coded the interviews, following standard methods of qualitative empirical research [196].

Threats to validity. Regarding external validity, our study may suffer from a selection bias, as common for these kinds of studies. Many of our participants work on 3D printers, which may have different characteristics. However, overall we reached developers from several different domains and did not observe any systematic differences. Finally, we focus on open source whereas results may differ in industrial settings in which forks are centrally managed.

Regarding internal validity, communication issues may have affected some answers; we mitigated this threat by refining our interview guide when questions raised confusion and involved two researchers in each interview. Despite open-ended questions and careful design (see above), we cannot entirely exclude confirmation bias, in which participants might avoid raising critical points; we mitigate this by focusing on insights gained, not just claims.

Results. Regarding RQ3 (clustering quality), participants mostly confirmed that the clustering results were appropriate, but often fine-tuned them with further splitting and joining. This further supports the need for interactive tools. Overall, participants supported our decision to cluster changes in a fork. For example, participant P4 said: *"It is necessary to split code changes into pieces, even though they cannot be executed in isolation."* Of the 11 participants, 10 said that INFOX correctly identified the clusters most of the time, although there are small clusters (containing one or two lines) should have been merged into bigger clusters. The remaining participant pointed out a cluster containing unrelated code that was automatically generated by libraries and should be removed.

As we discussed earlier, INFOX provides flexibility to developers by allowing them to split or join clusters interactively. During the interviews, participants compared the splitting and joining results carefully, and after several steps, they usually identified clusters that they agreed with. For example a typical interaction flowed as follows, here from participant P5: *"I think this blue and yellow cluster should belong together.. [clicks the join button] ..oh, so your software correctly identifies all of this being one thing not two different things."*

The participants identified some cases in which the clustering result could be improved, usually caused by technical limitations of the dependency analysis in our prototype (see Section 8). For example, when P4 found a 1-line cluster that should belong to another bigger cluster, the participant said: *"I know it is related, acceleration and volumetric (are related), but looking at just the syntax it is not, it is not using the same words. Adding check-box to manually merge selected clusters (could solve this problem)"*.

In summary, **participants generally agreed that INFOX could identify correct clusters at certain splitting or joining steps** (RQ3). Participants suggested that INFOX could provide more flexibility for manually refining the clustering result. Even though limited to few participants, our interviews corroborate the high-accuracy results from our quantitative study in a realistic setting.

With regard to RQ4 (overview), we looked particularly for signs that developers learned new insights while exploring the overview. Of the 11 participants, we showed 10 participants (P2-P11) the overview of forks in their project and eight of them gained different kinds of new information from the overview page:

- *Finding redundant development.* Two participants found other forks that are working on the same feature implementation as they did before. When they found these instances of

redundant development, they explored the fork's source code. For example, P3 said :*"It does look like somebody did a very simple one-function [...] system. I think they should use our code, there is great reason to use it."* After skimming the overview page, P4 said: *"I can see multiple forks are working on the similar problem. This one looks like it is adding [...] that I already added."*

- *Find interesting and potentially reusable feature.* When skimming all the forks, 6 participants identified specific features of interest; For example, P5 expressed *"this is all laser stuff, this is useful."* When participants mentioned something is interesting, we asked them why. The answers all identify features that are important to the project or that they could reuse in their own forks, such as P5's statement *"If it is only exists in this fork, then I want to somehow get this fork into my fork."*

Beyond these specific actionable insights, many participants more generally indicated that this overview would be useful: By looking at the overview page, our participants found many forks that they did not know before, and by reading the summary table of each fork, they usually got the idea of what has happened in each fork. For example, participant P3 said: *"It is going to make it a lot easier to find the things you are looking for as a programmer."* and P6 explained *"I see all the differences for all the forks. Basically it is the same thing I am doing through GitHub, (but) only it is summarized in the same place, I don't have to jump and open 50 tabs to do it."* Participant P7 expressed interest to use the tool for another project he maintained, for which he always wanted to know what is going on in forks, but was limited by current tools.

Regarding labels for code they did not know, we could observe that they clearly gave some initial idea to participants and could typically describe what they would expect from the implementation. For example, participant P5 described *"the [keywords] give me some clues of temperature; I know which part of Smoothie is modified."* Overall, all participants thought the interpretation of keywords is similar to their understanding of the source code.

In summary, even though we interviewed only a small number of participants, **we found frequent and concrete evidence of new insights gained from the overview page, including redundant development and reusable contributions** (RQ4). This is encouraging for the usefulness of the approach and its capability to provide actionable insights.

5.5 Related Work

Understanding branches and forks. Conceptually closest to our work is Bird and Zimmerman's analysis of branches at Microsoft, revealing that too many branches can be an issue and *what-if* analysis to explore the costs of merging can support decision making [36]. In addition, several studies have studied forking practices in open source and industrial product line development [71, 153, 190, 214]. Those studies have revealed the discussed problems, but did not provide any solutions.

Untangling code changes. Technically, our work relates to work on untangling code changes. Originally, untangling code changes was driven by biases in mining repositories and predicting defects [67, 110]. Barnett et al. [23] proposed CLUSTERCHANGES to decompose tangled code changes in order to identify independent parts of changes, especially large commits, to facilitate

understanding during the code reviewing process. A key assumption is that commits are not always cohesive and reliable. These approaches often analyze dependencies within a change and our implementation was inspired by and improves upon CLUSTERCHANGES, as discussed and evaluated.

Other strategies have been explored to untangle changes, including *semantic history slicing* that compares test executions [134, 135], and *EpiceaUntangler* [67] and *Thresher* [217] which interact with developers when committing a change, to encourage more cohesive commits. All these approaches are less applicable in our setting, as they would require test cases for all added functionality or upfront clean commits by all developers. In fact, Herzig and Zeller [111] argue that tangled changes are natural and should not be forbidden; we support this view and build tooling that extracts features after the fact, but at much larger granularity of differences in forks.

Concern location. Concern location (or concept or feature location) is the challenge of identifying the parts of the source code that correspond to a specific functionality, typically for maintenance tasks [178]. Based on a keyword or entry-point, developers or tools attempt to identify all code relevant for that feature. Concern location typically uses either a static, a dynamic, or an information-retrieval strategy [68, 239]: Static analyses examine structural information such as control or data flow dependencies [50, 188], whereas dynamic analyses examine the system's execution [56, 77]. In contrast, information-retrieval-based analyses perform some sort of search based on keywords [53, 68, 94, 144, 170] with more or less sophisticated natural language processing [116, 207]. Combinations of these strategies are common [68]. Our analysis has similarities with static concern-location approaches, but the setting is different: Instead of identifying code related to a specific given code fragment in a single code base, we aim at dividing the difference between two snapshots into cohesive code fragments without starting points. Whereas location usually identifies one concern at a time, we identify multiple features in a fork. At the same time, if execution traces or external keywords were available, those could likely be integrated into a clustering process like INFOX.

Code summarization. Finally, there are many approaches to summarize source code [129, 158, 175, 213] using information retrieval to derive topics from the vocabulary usage at the source code level. So far, we use only a standard lightweight information-retrieval technique to identify keywords for clusters, but combinations with more advanced summarization strategies might improve results significantly.

5.6 Discussion

Evidence from both academia and industry shows that current fork-based development is popular but has many practical problems that can be traced to a lack of transparency. Because developers do not have an overview of forks of a project, problems like redundant development, lost contributions and suboptimal forking point arise. To improve the transparency, we designed an approach to identify features from forks and generate an overview of the project in order to inform developers of what has happened in each active fork.

INFOX is a first step in making transparent what happens in forks of a project, and it can be a building block in a larger endeavor to support fork-based development, such that it keeps its main benefits, such as ease of use and distributed and independent development, while addressing many of its shortcomings through tool support.

This new transparency, might address problems including lost contributions and redundant development. All participants in our human subject study had immediate ideas of who might benefit from such a tool, including *"the person who maintains the main branch"* [P4] and *"it is super useful for everybody, especially for major main Smoothieware developers"*[P6]. In addition our evaluation has shown that clustering results are accurate (90 % on average) and labels are meaningful summaries.

At the same time, INFOX is just an initial prototype with technical limitations and many opportunities for extensions:

- The initial clustering strategy as well as the community-detection algorithm [95] are designed to divide a change into disjoint clusters. Boundaries between features are not always easy to define and features may overlap or may be split into subfeatures. Exploring other network analysis techniques to identify overlapped features or sub-features is an interesting avenue for further research.
- Although our clustering approach achieved high accuracy results, it would be worth to explore additional information that might provide insights about relationships of code fragments (even if unreliable generally), such as data-flow dependencies, syntactic or structural similarity between code fragments, code fragments that have been changed together in the same commit or by the same author. To identify which of these provide useful insights and which just create more noise.
- While INFOX currently aims at supporting exploration and navigation by summarizing features, it lays a foundation for future interactive tool support that can refine and persist features (e.g., for a product-line platform [17, 39, 192]) and support developers in merging selective changes across forks (e.g., generating pull requests).

5.7 Productization: forks-insight.com

To increase the impact of INFOX, we built a more light-weight and accessible web service – **FORKS INSIGHT** (available at: `forks-insight.com`). The user interface is shown in Figure 5.9.

FORKS INSIGHT provides facilities to explore unintegrated changes to find opportunities for reuse, to find inspirations for further development, to potentially connect developers working on similar topics. It analyzes each active fork of a repository by taking the code change differences between the latest commit of both the upstream and the fork to get the commits that only exist in forks, and extracting keywords from corresponding code changes, comments and commit messages. Users could log in with their GITHUB account and follow the project they are interested and get an overview of each repository in fork-level granularity. Also, we support key words searching, which potentially could help to identify interesting features and redundant development. Besides, FORKS INSIGHT presents statistical data of unintegrated changes at different granularities, such as commits, changed files, lines of code.

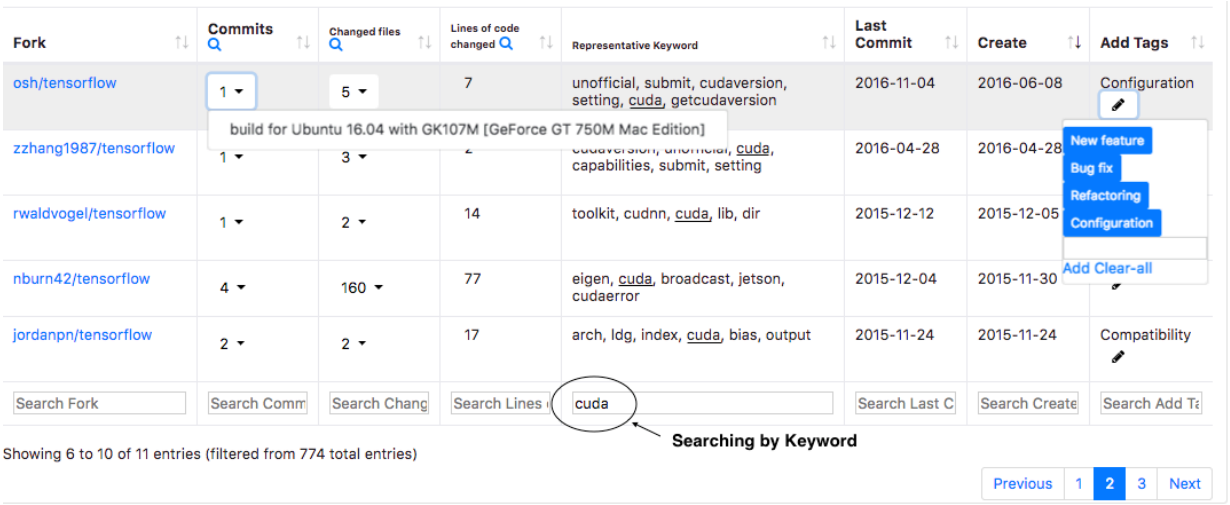


Figure 5.9: User Interface of FORKS INSIGHT. This example shows searching "cuda" in repository of *tensorflow/tensorflow*.

In addition, FORKS INSIGHT allows users to tag each fork based on their understanding of the main activity (see the last column in Figure 5.9). As developers fork a repository for different reasons: adding new features, fixing bugs, and changing configuration, etc [71, 153, 190, 214]. This information could help developers quickly find specific forks they want to explore. We hope user's input on tags could not only help themselves maintain and understand each fork, but also help the whole community, especially for the new users who are not familiar with this repository to get a better overview.

In order to improve the usability of FORKS INSIGHT, we plan to ask for feedback from open source developers. And we would like to add more interactive elements and powerful functions into our tool. There are several directions we are considering to move forward: using more visualization to show meaningful data; identifying features in forks; summarizing the activities of forks by natural language.

5.8 Summary

In this chapter, we described our first tooling intervention to improve the awareness of a community and generate a better overview of fork-based development mechanism. This is complementary to the solution described in Chapter 3 – identifying *natural interventions* that are correlated collaboration efficiency using fork-based development mechanisms.

To achieve our goal of generating a better overview of forks in a project, we design an approach INFOX to identify unmerged cohesive code changes (named features) from forks. To evaluate the effectiveness and usefulness of our tool, we designed both quantitative and qualitative study. To improve the research impact, we developed a lightweight, more user-friendly, and programming language independent web service *forks-insight.com*.

Chapter 6

New Intervention: Identifying Redundancies in Fork-based Development

This chapter shares material with the SANER'19 paper " Identifying Redundancies in Fork-based Development" [185].

In previous chapters, we described our solutions to mitigate collaboration inefficiencies when using fork-based development mechanisms: Identifying *natural interventions* (Chapter 3) and designing *new interventions* (Chapter 5). In this chapter, we demonstrate our second *new tooling intervention* design – using public information to address one of the inefficiencies emerged in fork-based development – redundant development as we described in Chapter 1.

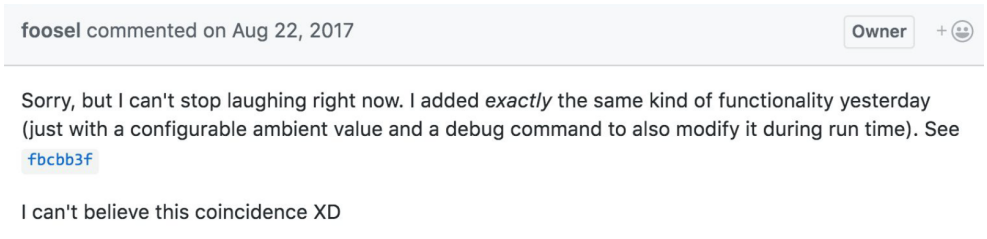
6.1 Motivation

As we described in Chapter 1, unaware of activities in other forks, developers may re-implement functionality already developed elsewhere. For example, Figure 6.1(a) shows two developers coincidentally working on the same functionality, where only one of the changes was integrated.[1] And Figure 6.1(b) shows another case in which multiple developers submitted pull requests to solve the same problem.[2]

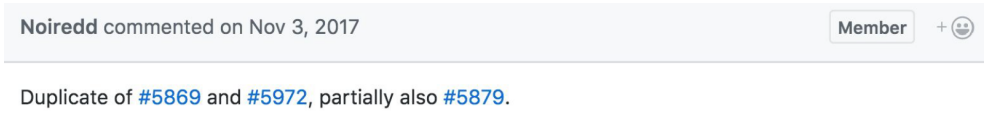
A developer we interviewed in the INFOX project (Chapter 5) [249] also confirmed the problem as follows: *"I think there are a lot of people who have done work twice, and coded in completely different coding style."* Gousios et al. [97] summarized nine reasons for rejected pull requests in 290 projects on GITHUB, in which 23% were rejected due to redundant development (either parallel development or superseded other pull requests). In analyzing the fraction of pull requests rejected due to redundancies over 1311 GITHUB projects (described in Chapter 1 and 3), we found that redundant development is a small but pervasive problem: about 1–5 % of all pull requests and 5–50 % of rejected pull requests (shown in Figures 1.3c).

Existing work shows that redundant development significantly increases the maintenance effort for maintainers [71, 214]. Specifically, Yu et al. manually studied pull requests from 26

[1] `https://github.com/foosel/OctoPrint/pull/2087`
[2] `https://github.com/BVLC/caffe/pull/6029`



(a) Two developers work on same functionality



(b) Multiple developers work on same functionality

Figure 6.1: *Pull requests rejected due to redundant development.*

popular projects on GITHUB, and found that on average 2.5 reviewers participated in the review discussions of redundant pull requests and 5.2 review comments were generated before the duplicate relation is identified [245]. Also, Steinmacher et al. [212] analyzed quasi-contributors whose contributions were rejected from 21 GITHUB projects and found that one-third of the developers declared the nonacceptance *demotivated* them from continuing to contribute to the project.

Facing this problem, the goal of our research is (1) to help project maintainers to automatically identify redundant pull request order to decrease the workload of reviewing redundant code changes, and (2) to help developers detect redundant development as early as possible by comparing code changes with other forks in order to eliminate wasted effort and encourage developers to collaborate.

To achieve our goal, we first identify clues that indicate a pair of code changes might be similar by manually checking 45 duplicate pull request pairs. Then we design measures to calculate the similarity between changes for each clue. Finally, we treat the list of similarities as features to train a classifier in order to predict whether a pair of changes is a duplicate (Research method is shown in Figure 6.3).

We evaluate the effectiveness of our approach from different perspectives, which align with the application scenarios introduced before for our bot: (1) helping project maintainers to identify redundant pull requests in order to decrease the code reviewing workload, (2) helping developers to identify redundant code changes implemented in other forks in order to save the development effort, and encouraging collaboration. In these scenarios, we prefer high precision and could live with moderate recall, because our goal is to save maintainers' and developers' effort instead of sending too many false warnings. Sadowski et al. found that if a tool wastes developer time with false positives and low-priority issues, developers will lose faith and ignore results [194]. We argue that as long as we show some duplicates without too much noise, we think it is a valuable addition. The result shows that our approach could achieve 57–83% precision for identifying duplicate pull requests from the maintainer's perspectives within a reasonable threshold range

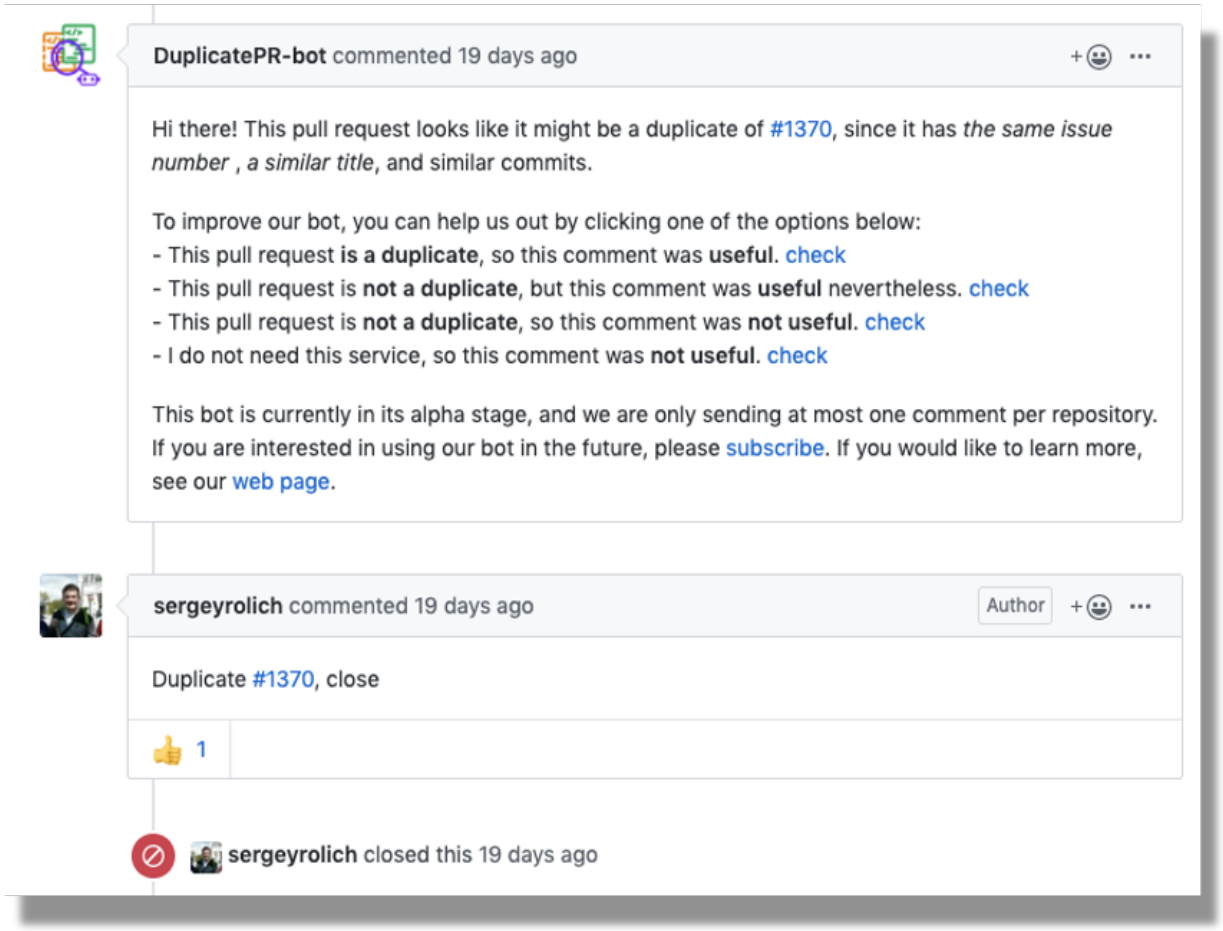


Figure 6.2: Duplicate Pull Request Detector: A GitHub Bot

(details in Section 6.5). Also, our approach could help developers save 1.9–3.0 commits per pull request on average . We also compared our approach to the state-of-the-art showing that we could outperform the state-of-the-art by 16–21% recall. Finally, we conducted a sensitive analysis to investigate how sensitive our approach is to different kinds of clues in the classifier.

6.2 Application Scenarios

Our approach could be applied to different scenarios to help different users. Primarily, we envision **a** GitHub **bot** (shown in Figure 6.2) to monitor the incoming pull request in a repository and compare the new pull request with all the existing pull requests in order to *help project maintainers* to decrease their workload. The bot would automatically send warnings when a duplication is detected to send warnings when duplicate development is detected, and this could assist maintainers' and contributors' work in open source projects as Wessel et al. described [236].

In addition to monitor incoming pull request for each project, we envision **a bot to monitor forks and branches**, and compare the commits with other forks and with existing pull requests,

in order to *help developers* detect early duplicate development. Researchers have found that developers think it is worth spending time checking for existing work to avoid redundant development, but once they start coding a pull request, they never or rarely communicate the intended changes to the core team [100]. We believe it is useful to inform developers when potentially duplicate implementation is happening in other forks, and encourage developers to collaborate as early as possible instead of competing after submitting the pull request. Also, we could build a plug-in for a development IDE, so we could detect redundant development in real time.

6.3 Research Method

To achieve our goal, we design a 4-step research method (shown in Figure 6.3). First, we identify clues that indicate a pair of code changes might be similar by manually checking 45 duplicate pull request pairs. Then we design measures to calculate the similarity between changes for each clue. Finally, we treat the list of similarities as features to train a classifier in order to predict whether a pair of changes is a duplicate. Our dataset and the source code are available online.[3]

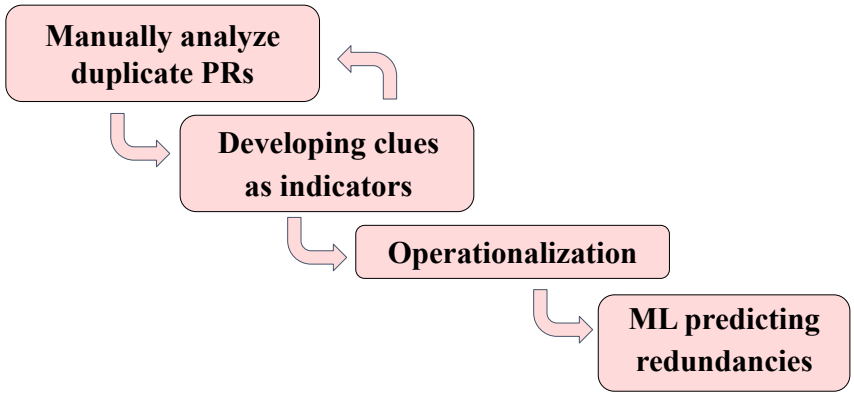


Figure 6.3: Research Method of Identifying Redundancies in Forks.

6.3.1 Identifying Clues to Detect Redundant Changes

We refer to *changes* when developers make code changes in a project. There are different granularities of changes, such as pull requests, commits, or fine-grained code changes in the IDE (Integrated Development Evironment). In this section, we show how we extracted clues that indicate the similarity between pull requests. Although we use pull requests to demonstrate the problem, note that the examples and our approach are generalizable to different granularities of changes: For example, we could detect redundant pull requests for an upstream repository, redundant commits in branches or forks, or redundant code changes in IDEs (detailed application scenarios are described in Sec. 6.2).

[3] `https://github.com/shuiblue/INTRUDE-refactor`

Next, we present two motivating examples of duplicate pull requests from GITHUB to motivate the need for using both natural language and source code related information in redundant development detection.

Case 1: Similar Text Description but Different Code Changes

We show a pair of duplicate pull requests that are both fixing the bug 828266 in the *mozilla-b2g/gaia* repository in Figure 6.4. Both titles contained the bug number, copied the title of the bug report, and both descriptions contain the link to the bug report. It is straightforward to detect duplication by comparing the referred bug number, and calculating the similarity of the title and the description. However, if we check the source code,[4] the solutions for fixing this bug are different, although they share two changed files. Maintainers would likely benefit from automatic detection of such duplicates, even if they don't refer to a common bug report. It could also prevent contributors from submitting reports that are duplicates, lowering the maintenance effort.

Case 2: Similar Code Changes but Different Text Description

We show a pair of duplicate pull requests that implement similar functionality for project *mozilla-b2g/gaia* in Figure 6.5. Both titles share words like *'Restart(ing)'* and *'B2G/b2g'*, and both did not include any other textual description beyond the title. Although one pull request mentioned the bug number, it is hard to tell whether these two pull requests are solving the same problem by comparing the titles. However, if we include the code change information, it is easier to find the common part of the two pull requests: They share two changed files, and the code changes are not identical but very similar except the comments at Line 8 and the code structure. Also, they changed the code in similar locations.

6.3.2 Clues for Duplicate Changes

We might consider existing techniques for clone detection [27], which aim to find pieces of code of high textual similarity on Type 1-3 clones in a system but not textual descriptions [191]. However, our goal is not to find code blocks originating from copy-paste activities, but code changes written independently by different developers about the same functionality due to a lack of an overview in the fork-based development environment, which is conceptually close to the *Type-4 clones* [191], meaning two code changes have function similarity but they are different in syntax.

Similarly, we have considered existing techniques for detecting duplicate bug reports [120, 142, 174, 208, 233, 248], which compare textual descriptions but not source code. Different from the scenarios of clone detection and detecting duplicate bug reports, for detecting duplicate pull requests we have both textual description and source code, including information about changed files and code change locations. Thus we have additional information that we can exploit, and have opportunities to detect duplicate changes more precisely. Therefore, we seek inspiration

[4] `https://github.com/mozilla-b2g/gaia/pull/7587/files`
`https://github.com/mozilla-b2g/gaia/pull/7669/files`

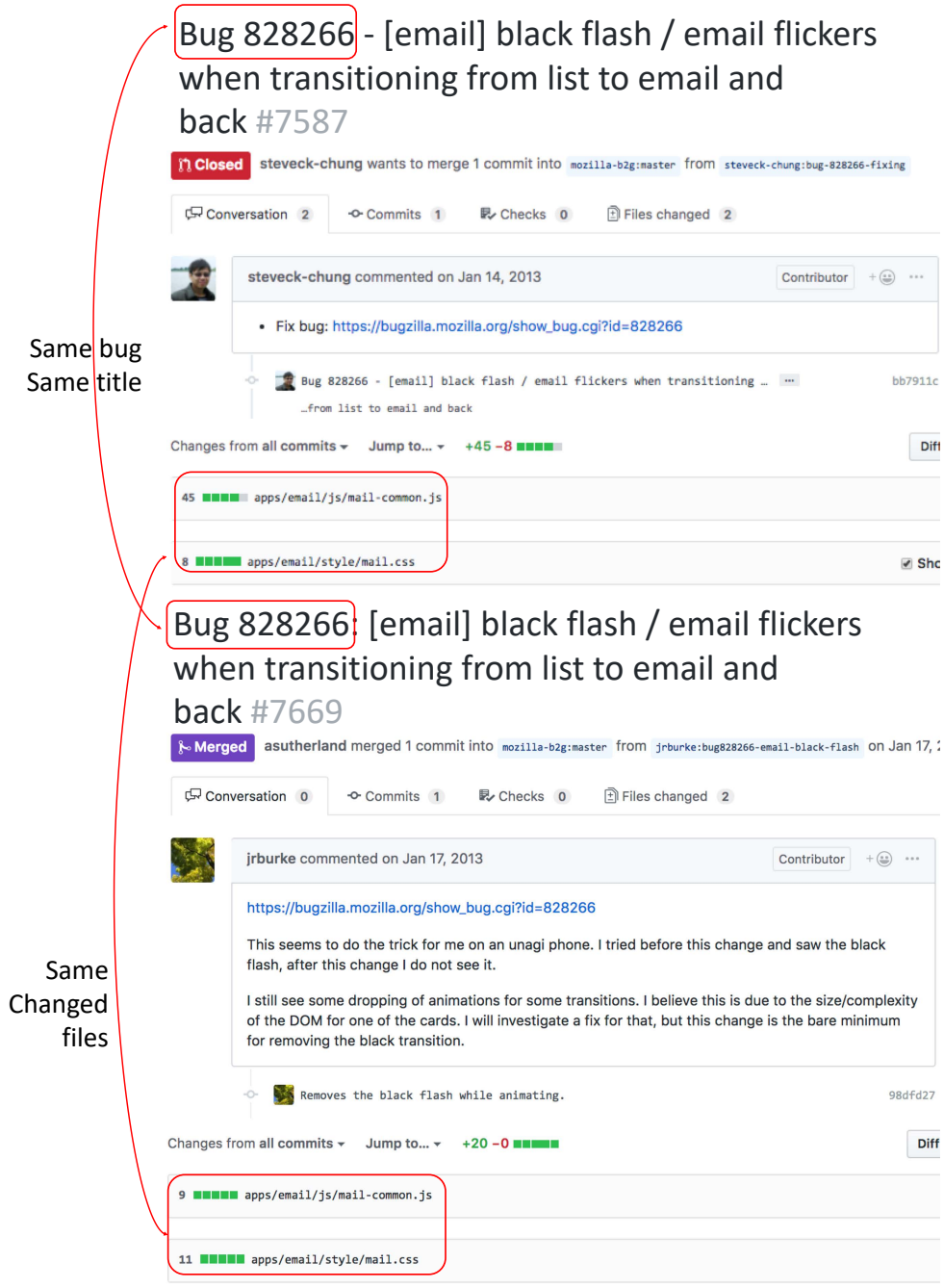


Figure 6.4: *Screenshot - Duplicate pull requests with similar text information*

Figure 6.5: *Screenshot - Duplicate pull requests with similar code change information*

from both lines of research, but tailor an approach to address the specific problem of detecting redundant code changes across programming languages and at scale.

To identify potential clues that might help us to detect if two changes are duplicates, we randomly sampled 45 pull requests that have been labeled as *duplicate* on GITHUB from five projects using (the March 2018 version of) GHTorrent [96]. For each, we manually search for the corresponding pull request that the current pull request is duplicate with. We then went through each pair of duplicate pull requests and inspected the text and code change information to extract clues indicating the potential duplication. We iteratively refined the clues until analyzing more duplicate pairs yielded no further clues.

Based on this manual inspection result, neither text information or code change information was always superior to the other in all cases. Text information represents the external goal and summary of the changes by developers, while the corresponding code change information explicitly describes the internal behavior. Thus, using both kinds of information can make it possible to detect redundant development precisely. Comparing to previous work [136], which detects duplicate pull requests by calculating the similarity only of title and description, our approach considers multiple facets of both the text information and the code change information.

We summarize the clues characterizing the content of a code change, which we will use to calculate change similarity:

- **Change description** is a summary of the code changes written in natural language. For example, a commit has commit messages and a pull request contains title and description. Similar titles lead to a strong indicator that these two code changes are solving a similar problem. The description contains more detailed information of what kind of issue the current code changes are addressing, and how. If the descriptions of the two code changes are similar in terms of meaningful keywords, there is a higher chance that they are implementing the same functionality. However, textual description alone might be insufficient, as Figure 6.4 shows.
- **Reference to issue tracker** is a common practice that developers explicitly link the code change to an existing issue or feature request in the issue tracker (as shown in Figure 6.4. If both code changes reference the same issue, it is likely redundant, except for cases in which the two developers intended to have two solutions to further compare.
- **Patch content** is the differences of text changes in each file by running *'git diff'* command. The content could be source code written in different programming languages or comments from source code files or could be plain text from non-code files. We found (when inspecting redundant development) that developers often share keywords when defining variables and functions so that extracting representative keywords from each patch could help us identify redundant changes more precisely compared to only using textual description (as shown in Figure 6.5).
- **A list of changed files** contains all the changed files in the patch. We assume that if two patches share the same changed files, there is a high chance that they are working on similar or related functionality. For example, in Figure 6.5, both pull requests changed the *helper.js* and *perf.js* files.
- **Code change location** is a range of changed lines in the corresponding changed files. If the code change location of two patches are overlapping, there is a potential that they are redundant. For example, in Figure 6.5, two pull requests are both modifying *helper.js* lines

Table 6.1: Clues and corresponding machine learning features

Clue	Feature for Classifier	Value
Change description	Title_similarity	[0,1]
	Description_similarity	[0,1]
Patch content	Patch_content_similarity	[0,1]
	Patch_content_similarity_on_ overlapping_changed_files	[0,1]
Changed files list	Changed_files_similarity	[0,1]
	#Overlapping_changed_files	N
Location of code changes	Location_similarity	[0,1]
	Location_similarity_ on_overlapping_changed_files	[0,1]
Reference to issue tracker	Reference_to_issue_tracker	{-1, 0, -1}

8–22, which increases the chance that the changes might be redundant.

6.4 Identifying Duplicate Changes in Forks

Our approach consists of two steps: (1) calculating the similarity between a pair of changes for each clue listed previously; (2) predicting the probability of two code changes being duplicate through a classification model using the similarities of each clue as features.

As our goal is to find duplicate development caused by unawareness of activities in other forks, we first need to filter out pull request pairs in which the authors are aware of the existence of another similar code change by checking the following criteria:

- Code changes are made by the same author; or
- Changes from different authors are linked on GITHUB by authors, typically used when one is a following work of the other, or one is intended to supersede the other with a better solution; or
- The later pull request is modifying the code on top of the earlier merged pull request.

6.4.1 Calculating Similarities for Each Clue

We calculate the similarity of each clue as features to train the machine learning model. Table 6.1 lists the features.

Change Description. To compare the similarity of the description of two changes, we first preprocess the text through tokenization and stemming. Then we use the well-known *Term Frequency Inverse Document Frequency* (TF-IDF) scoring technique to represent the importance of each token (its TF-IDF score), which increases proportionally to the number of times a word appears in the feature's corpus but is offset by the frequency of the word in the other feature's

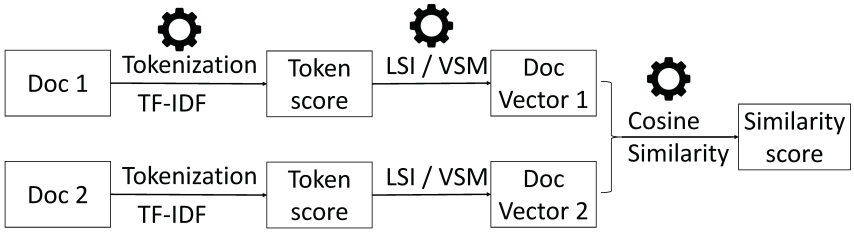


Figure 6.6: Calculating similarity for description / patch content

corpora [197]. The TF-IDF score reflects the importance of a token to a document; tokens with higher TF-IDF values better represent the content of the document. For example, in Figure 6.5, the word *'LockScreen'* appears many times in both pull requests, but does not appear very often in the other parts of the project, so the *'LockScreen'* has a high TF-IDF score for these pull requests.

Next, we use Latent Semantic Indexing (LSI) [130] to calculate similarity between two groups of tokens, which is a standard natural language processing technique and has been proved to outperform other similar algorithms on textual artifacts in software engineering tasks [49, 177]. Last, we calculate the cosine similarity of two groups of tokens to get a similarity score (see Figure 6.6).

Patch Content. We compute the token-based difference between the previous and current version of the file of each change, e.g. if original code is *func(argc1, argc2)*, and updated version is *func(argc1, argc2, argc3)*, we only extract *argc3* as the code change. We do not distinguish source code, in-line comments, and documentation files, we treat them uniformly as source code, but assume the largest portion is source code.

In order to make our approach programming languages independent, we treat all source code as text. So we use the same process as code change description to calculate the similarity, except we replace LSI by Vector Space Model (VSM), shown in Figure 6.6, because VSM works better in case of exact matches while LSI retrieves relevant documents based on the semantic similarity [49].

However, this measure has limitations. When a pull request is duplicate with only a subset of code changes in another pull request, the similarity between these two is small, which makes it harder to detect duplicate code changes. During the process of manually inspecting duplicate pull request pairs (Section 6.3.1), we found there are 28.5% pairs where one pull request is five times larger than the other at the file level. To solve this problem, we add another feature as the similarity of patch content only on overlapping files.

Changed Files List. We operationalize the similarity of two lists of files into computing the overlap between two sets of files by using Jaccard similarity coefficient: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ (A and B are two sets of elements). The more overlapping files two changes have, the more similar they are. As Figure 6.7 shows, PR1 and PR2 have modified 2 files each, and both of them modified *File1*, so the similarity of the two lists of files is $1/3$.

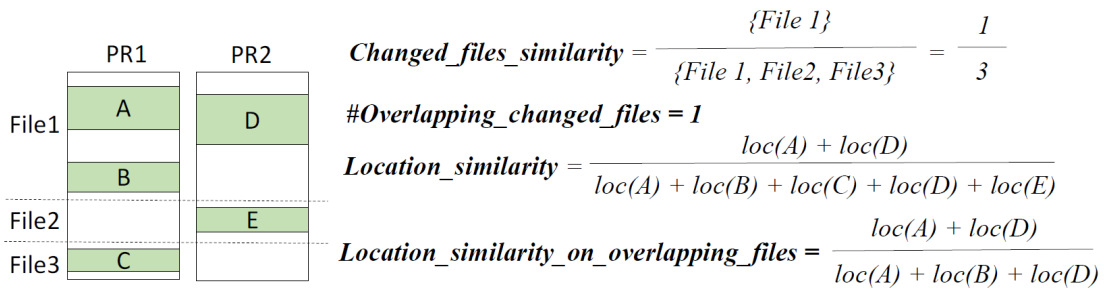


Figure 6.7: *Similarity of changed files and code change location (loc: Lines of code).*

Again, in case that one pull request is much bigger than the other in terms of changed files, which leads to a small ratio of overlapping files, we add a feature defined as the number of overlapping files.

Location of Code Changes We calculate the similarity of code change location by comparing the size of overlapping code blocks between a pair of changes. The more overlapping blocks they have, the more similar these two changes are. In Figure 6.7, block *A* overlaps with block *D* in *File1*. We define the *Location similarity* as the length of overlapping blocks divided by length of all the blocks.

Similar to our previous concern, in order to catch redundant changes between big and small size of patches in file level, we define a feature of similarity of code change location for only overlapping files. For example, in Figure 6.7, block *A*, *B* and *D* belong to *File1*, but block *C* and *E* belong to different files, so the measure of *Location similarity on overlapping changed files* only consider the length of block *A*, *B* and *D*.

Reference to Issue Tracker. Based on our observation, we found that if two changes link to the same issue, they are very likely duplicates, while, if they link to different issues, our intuition is that the chance of the changes to be duplicate is very low. So we defined a feature as reference to issue tracker. For projects using the GITHUB issue tracker, we use the GITHUB API to extract the issue link, and, for projects using other issue tracking systems (as Figure 6.4 shows), we parse the text for occurrences from a list of patterns, such as 'BUG', 'ISSUE','FR' (short for feature request). We define three possible values for this feature: If they link to the same issue, the value is 1; if they link to different issues, the value is -1; otherwise it is 0.

6.4.2 Predicting Duplicate Changes Using Machine Learning

The goal is to classify a pair of changes as duplicate or not. We want to aggregate these nine features and make a decision. Since it is not obvious how to aggregate and weigh the features, we use machine learning to train a model. There are many studies addressing the use of different machine learning algorithms for classification tasks, such as support vector machines, AdaBoost, logistic regressions, neural network, decision Trees, random forest, and k-Nearest Neighbor [121]. In this study, in order to assess the performance of these techniques for our redundancy detection problem, we have conducted a preliminary experimental study. More specifically, we compared

the performance of six algorithms based on a small set of subject projects. We observed that the best results were obtained when AdaBoost [88] was used. Therefore, we focused our efforts only on AdaBoost, but other techniques could be easily substituted. Since the output of AdaBoost algorithm is a probability score whether two changes are duplicate, we set a threshold and report two changes as duplicate when the probability score is above the threshold.

6.5 Evaluation: Effectiveness

We evaluate the effectiveness of our approach from different perspectives, which align with the application scenarios introduced in Sec. 6.2: (1) helping project maintainers to identify redundant pull requests in order to decrease the code reviewing workload, (2) helping developers to identify redundant code changes implemented in other forks in order to save development effort. To demonstrate the benefit of incorportating multiple clues, we compared our approach to the state-of-the-art that uses textual comparison only. Finally, beyond just demonstrating that our specific implementation works, we explore the relative importance of our clues with a sensitivity analysis, which can guide other implementations and future optimizations. Thus, we derived four research questions:

- *RQ1: How accurate is our approach to help maintainers identify redundant pull requests?*
- *RQ2: How much effort could our approach save for developers in terms of commits?*
- *RQ3: How good is our approach identifying redundant pull requests comparing to the state-of-the-art?*
- *RQ4: Which clues are important to detect duplicate changes?*

6.5.1 Dataset

To evaluate approaches, an established ground truth is needed—a reliable data set defining which changes are duplicate. In our experiment, we used an established corpus named DupPR, which contains 2323 pairs of duplicate pull requests from 26 popular repositories on GITHUB [245] (Table. 6.2). We picked half of the Duppull request dataset, which contains 1174 pairs of duplicate pull requests in twelve repositories as the *positive samples* in the training dataset (highlighted) to calibrate our classifier (see Sec. 6.4.2), and the remaining 1149 pairs from 14 repositories as testing dataset.

While this dataset provides examples of duplicate pull requests, it does not provide negative cases of pull request pairs that are not redundant (which are much more common in practice [97]). To that end, we randomly sampled pairs of merged pull requests from the same repositories, as we assume that if two pull requests are both merged, they are most likely not duplicate. Overall, we collected 100,000 negative samples from the same projects, 50,000 for training, and 50,000 for testing.

Table 6.2: subject projects and their duplicate PR pairs.

Repository	#Forks	#PRs	#DupPR pairs	Language
symfony/symfony	6446	16920	216	PHP
kubernetes/kubernetes	14701	38500	213	Go
twbs/bootstrap	62492	8984	127	CSS
rust-lang/rust	5222	26497	107	Rust
nodejs/node	11538	12828	104	JavaScript
symfony/symfony-docs	3684	7654	100	PHP
scikit-learn/scikit-learn	15315	6116	68	Python
zendframework/zendframework	2937	5632	53	PHP
servo/servo	1966	12761	52	Rust
pandas-dev/pandas	6590	9112	49	Python
saltstack/salt	4325	29659	47	Python
mozilla-b2g/gaia	2426	31577	38	JavaScript
rails/rails	16602	21751	199	Ruby
joomla/joomla-cms	2768	13974	152	PHP
angular/angular.js	29025	7773	112	JavaScript
ceph/ceph	2683	24456	104	C++
ansible/ansible	13047	24348	103	Python
facebook/react	20225	6978	74	JavaScript
elastic/elasticsearch	11859	15364	62	Java
docker/docker	14732	18837	61	Go
cocos2d/cocos2d-x	6587	14736	57	C++
django/django	15821	10178	55	Python
hashicorp/terraform	4160	8078	52	Go
emberjs/ember.js	4041	7555	46	JavaScript
JuliaLang/julia	3002	14556	42	Julia
dotnet/corefx	4369	17663	30	C#

* The upper half projects (highlighted) are used as training dataset,
and lower half projects are used as testing dataset.

6.5.2 Analysis and Results

RQ1: How accurate is our approach to help maintainers identify redundant contributions?

In our main scenario, we would like to notify maintainers when a new pull request is duplicate with existing pull requests, in order to decrease their workload of reviewing redundant code changes (e.g., a bot for duplicate pull request monitoring). So we simulate the pull request history of a given repository, compare the newest pull request with all the prior pull requests, and use our classifier to detect duplication: If we detect duplication, we report the corresponding pull request number.

Research method. We use the evaluation set of the Duppull request dataset as ground truth. However, based on our manual inspection, we found the dataset is incomplete, which means it

Table 6.3: RQ1: Simulating PR history

PR history	Our_result	DupPR	Manual checking	Warning correctness
1	-	?		
2	-	?		
3	-	?		
4	2	2		✓
5	-	?		
6	5	?	5	✓
7	-	?		
8	-	6		
9	4	?	✗	✗

Ground Truth

does not cover all the duplicate pull requests for each project. This leads to several problems. First, when our approach detects a duplication but the Duppull request does not cover the case, the precision value is distorted. To address this problem, we decided to manually check the correctness of the duplication warnings; in another word, we complement Duppull request with manual checking result as ground truth (shown in Table 6.3). Second, it is unrealistic to manually identify all the missing pull request pairs in each repository, so we decided to randomly sample 400 pull requests from each repository for computing precision.

Table 6.3 illustrates our replay process. The *PR_history* column shows the sequence of the coming pull requests, *our_result* column is our prediction result, for example, we predict 4 is duplicate with 2, and 6 is duplicate with 5, and 9 is duplicate with 4; *DupPR* column shows that 2 and 4 are duplicate, and 8 and 6 are duplicate; the *manual_checking* column shows that the first 2 authors manually checked and confirmed 5 and 6 are duplicate, and 9 and 4 are not duplicate. The warning correctness shows that the precision of this example is $2/3$.

For calculating recall, we use a different dataset because even for 400 pull requests per project, we need to manually check a large number of pull requests in order to find all the duplicate pull request pairs, which is very labor intensive. Thus, we only use the evaluation section of the Duppull request dataset (lower half of Table. 6.2) to run the experiment, which contains 1149 pairs of confirmed duplicate pull requests from 14 repositories.

Result. Figure 6.8 shows the precision and recall at different thresholds. We argue that within a reasonable threshold range of 0.5925–0.62, our approach achieved 57-83% precision and 10-22% recall. After some experiments, we pick a reasonable default threshold of 0.6175, where our approach achieves 83% precision and 11% recall (dash line in Figure 6.8). Tables 6.4 and 6.5 show the corresponding precision and recall for each the project separately at the default threshold.

We did not calculate the precision for lower threshold because when the threshold gets lower, the manual check effort becomes infeasible. Here we argue that a higher precision is more important than recall in this scenario, because our goal is to decrease the workload of maintainer, so that we hope all the warnings that we send to them are mostly correct, otherwise, we will waste

Table 6.4: RQ1, precision at default threshold

Repository	TP / TP + FP	Precision
django/django	5 / 5	100%
facebook/react	3 / 3	100%
hashicorp/terraform	3 / 3	100%
ansible/ansible	2 / 2	100%
ceph/ceph	2 / 2	100%
joomla/joomla-cms	2 / 2	100%
docker/docker	1 / 1	100%
cocos2d/cocos2d-x	6 / 7	86%
rails/rails	5 / 6	83%
angular/angular.js	3 / 4	75%
dotnet/corefx	2 / 3	67%
emberjs/ember.js	2 / 4	50%
elastic/elasticsearch	1 / 2	50%
JuliaLang/julia	1 / 2	50%
Overall	38 / 46	83%

their time to check false positives. In the future, it would be interesting to interview stakeholders or design experiment with real intervention to see acceptable levels about the acceptance rate of false positives in the real scenario, so we could allow users to set the threshold for different tolerance rate of false positives.

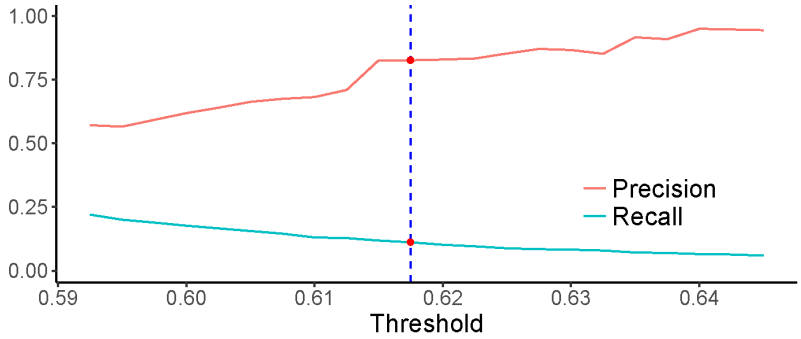


Figure 6.8: RQ1: Precision & Recall at different thresholds, dashed line shows the default threshold

RQ2: How much effort could our approach save for developers in terms of commits?

The second scenario focuses on developers. We would like to detect redundant development as early as possible to help reduce the development effort. A hypothetical bot monitors forks and branches and compares un-merged code changes in forks against pending pull requests and against code changes in other forks.

Table 6.5: RQ1, recall at default threshold

Repository	TP / TP + FN	Recall
ceph/ceph	31 / 104	30%
django/django	14 / 55	25%
hashicorp/terraform	8 / 52	15%
elastic/elasticsearch	7 / 62	11%
cocos2d/cocos2d-x	6 / 57	11%
rails/rails	20 / 199	10%
docker/docker	6 / 61	10%
angular/angular.js	11 / 112	10%
joomla/joomla-cms	12 / 152	8%
ansible/ansible	7 / 103	7%
emberjs/ember.js	3 / 46	7%
facebook/react	3 / 74	4%
JuliaLang/julia	0 / 42	0%
dotnet/corefx	0 / 30	0%
Overall	128 / 1149	11%

Research Method. To simulate this scenario, we replay the commit history of a pair of duplicate pull requests. As shown in Figure 6.9, when there is a new commit submitted, we use the trained classifier to predict if the two groups of existing commits from each pull request are duplicate.

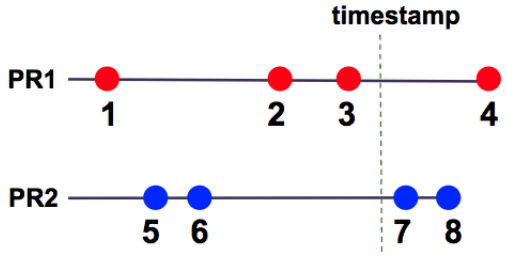
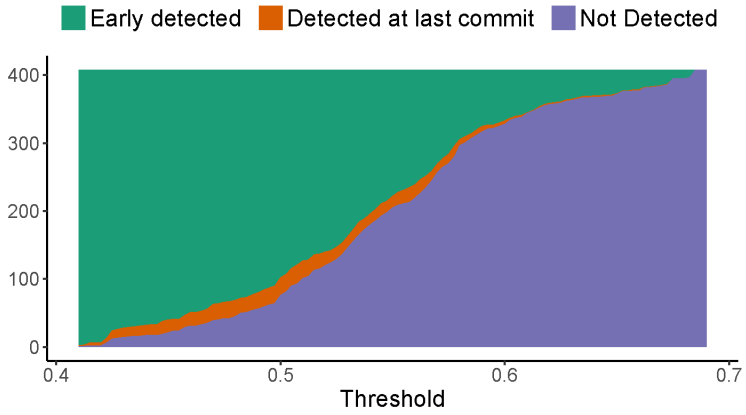
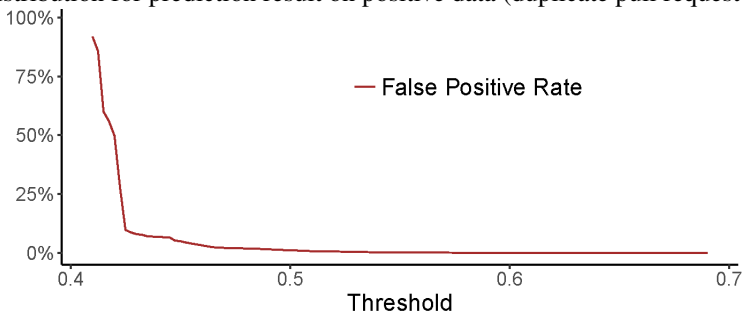


Figure 6.9: Simulating commit history of a pair of pull requests. If PR1 is duplicate with PR2, we first compare commit 1 and 5, if we do not detect duplication, then we compare 1 and (5,6), and so on. If we detect duplication when comparing (1, 2, 3) with (5, 6), then we conclude that we could save developers of PR1 one commit of effort or PR2 two commits.

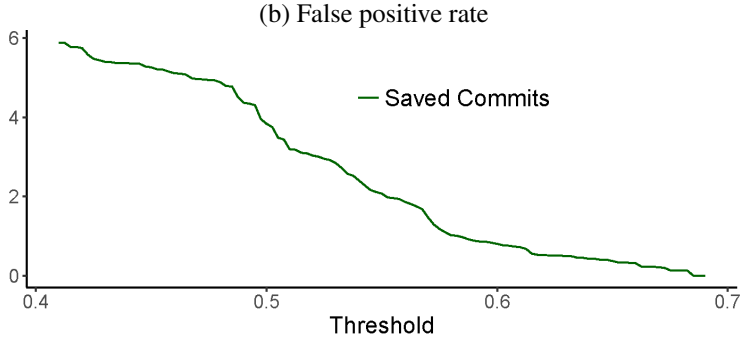
We use the same testing dataset as described in Sec. 6.5.1). We calculate the number of commits to represent the saved development effort because number of commits and lines of added/modified code are highly correlated [229]. Since we are checking if our approach could save developers' effort in terms of commits, we first need to filter out pull request pairs that have no chance to predict the duplication early. For instance, two pull requests both contain only one commit, or the later pull request has only one commit. After this filtering, the final dataset contains 408 positive samples and 13,365 negative samples.



(a) Distribution for prediction result on positive data (duplicate pull request pairs)



(b) False positive rate



(c) Saved #commits per pull request

Figure 6.10: RQ2: Can we detect duplication early, how much effort could we save in terms of commits, and corresponding false positive rate at different threshold

Result. Based on the classification result, we group the pairs of duplicate pull requests (positive dataset) into three groups: Duplicate detected early, duplication detected in the last commit, and duplication not detected. In addition, we check how much noise our approach introduces, we calculate the number of false positive cases among all the 13,365 negative cases, and get the false positive rate.

We argue that within a reasonable threshold range of 0.52–0.56, our approach achieved 46–71% recall (see Figure 6.10(a)), with 0.07–0.5% false positive rate (see Figure 6.10(b)). Also, we could save 1.9–3.0 commits per pull request within the same threshold range (see

Figure 6.10(c)).[5]

RQ3: How good is our approach identifying redundant pull requests comparing to the state-of-the-art?

Research Method. Yu et al. proposed an approach to detect duplicate pull requests with the same scenario as we described in RQ1 [136], that is, for a given pull request, identifying duplicate pull requests among other history pull requests. However, there are three main differences between their approach and ours: (1) they calculate the textual similarity between a pair of pull requests only on title and description, while we consider patch content, changed files, code change location, and reference to issue tracking system when calculating similarities (9 features) (see Sec. 6.3.1); (2) their approach returns top-K duplicate pull requests among existing pull requests by ranking them by arithmetic average of the two similarity values, while our approach reports duplication warnings only when the similarity between two pull requests is above a threshold; (3) they get the similarity of two pull requests by calculating the arithmetic average of the two similarity values, while we adopt a machine learning algorithm to aggregate nine features.

We argue that for this scenario, our goal is to decrease maintainers' workload for reviewing duplicate pull requests, instead of assuming maintainers periodically to go through a list of potential duplicate pull request pairs. In our solution, we therefore also prefer high precision over recall. But in order to make our approach comparable, we reproduced their experimental setup and reimplemented their approach, even though it does not align with our goal.

Research Method. We follow their evaluation process by computing *recall-rate@k*, as per the following definition:

$$\textit{recall-rate@}k = \frac{N_{\text{detected}}}{N_{\text{total}}} \tag{6.1}$$

N_{detected} is the number of pull requests whose corresponding duplicate one is detected in the candidate list of top-k pull requests, N_{total} is the total number of pairs of duplicate pull requests for testing. It is the ratio of the number of correctly retrieved duplicates divided by the total number of actual duplicates. The value of k may vary from 1 to 30, meaning the potential k duplicates.

Result. As shown in Figure 6.11, our approach achieves better results than the state-of-the-art by 16–21% *recall-rate@k*. The reason is that we considered more features and code change information when comparing the similarity between two changes. Also, we use a machine learning technique to classify based on features.

RQ4: Which clues are important to detect duplicate changes?

We aim to understand the clues that influence the effectiveness of our approach. Specifically, we investigate how sensitive our approach is to different kinds of clues in the classifier.

[5]Comparing to RQ1 scenario, we set a lower default threshold in this case, and we argue that developers of forks are more willing to inspect activities in other forks [100, 249]. But again, in the future, we would give developers the flexibility to decide how many notifications they would like to receive.

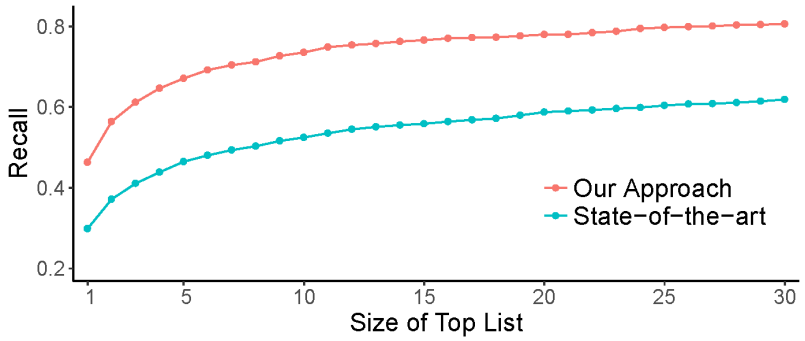


Figure 6.11: RQ3: How good is our approach identifying redundant pull requests comparing to the state-of-the-art?

Research Method. We design this experiment on the same scenario as RQ1, which is helping maintainers to detect duplication by comparing new pull request with existing pull requests from each project as testing dataset (see Sec. 6.5.2). However, we used a smaller testing dataset of 60 randomly sampled pull requests, because for calculating precision we need to manually check the detected duplicate pull request pairs every time, which is labor intensive.

We trained the classifier five times, and we removed one clue each time. So the combined absolute values of features change every time in the classifier's sum. This means that using a single cut-off thresholds for all the rounds does not make sense – the measured objective function changes all the time. Therefore, we pick the threshold for each model such that it produces a given recall (20%) and compare precision at that threshold.

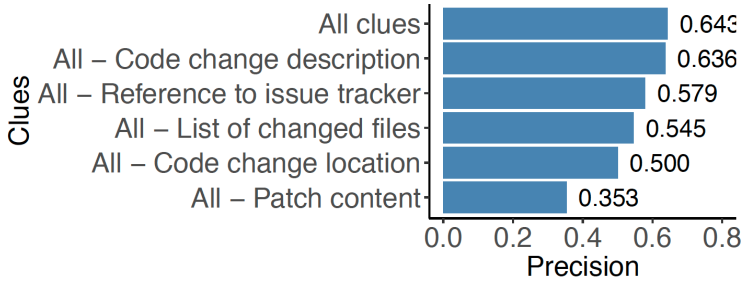


Figure 6.12: RQ4: Sensitive analysis, removing one clue at a time. Precision at recall fixed at 20%

Result. Figure 6.12 shows that when considering all the clues, the precision is the highest (64.3%). Removing the clue of patch content affects precision the most, which leads to 35.3% precision, and removing the text description has the least effect (63.6% precision). The result shows that patch content is the most important clue in our classifier, which likely explains the improvement in RQ3 as well. In the future, we could also check the sensitivity for each feature, or different combinations.

6.6 Related Work

Duplicate Pull Request Detection. Li et al. proposed an approach to detect duplicate pull requests by calculating the similarity on title and description [136], which we used as baseline to compare with (Sec 6.5.2). Different from their approach, we considered both textual and source code information, used a machine learning technique classify duplicates, and evaluated our approach in scenarios from the maintainer's and developer's perspectives. Later, Yu et al. created a dataset of duplicate pull request pairs [245], which we have used as part of our ground truth data.

Zhang et al. analyzed pull requests with a different goal: They focused on competing pull requests that edited the same lines of code, which would potentially lead to merge conflicts [241], which is roughly in line with the merge conflicts prediction tool Palantír [202] and Crystal [43]. Even though we also look at change location, we focus on a broader picture: We detect redundant (not only conflicting) work, and encourage early collaboration. In the future, we could report conflicts since we already collect the corresponding data as one feature in the machine learning model.

Duplicate Bug Report Detection. We focus on detecting duplicate pull requests, but there have been other techniques to detect other forms of duplicate submissions, including bug reports [29, 114, 174, 193, 208, 233] and StackOverflow questions [16]. On the surface, they are similar because they compare text information, but the types of text information is different. Zhang et al. [246] summarized related work on duplicate-bug-report detection. Basically, existing approaches are using information retrieval to parse different types of resource separately, such as natural-language [114, 193, 216], execution information [174, 208]. Further, Wang et al. [233] combined execution information with natural language information to improve the precision of the detection. Beyond information retrieval, duplicate bug-report classification [119] and the Learn To Rank approach were also used in duplicate detection [141, 247]. In contrast, our approach focuses on duplicate implementations for features or bug fixing, where we can take the source code into account.

Clone Detection. Our work is similar to the scenario of detecting *Type-4* clones: Two or more code fragments that perform the same computation but are implemented by different syntactic variants [191]. We, instead, focus on detecting work of independent developers on the same feature or bug fix, which is a different, somewhat more relaxed and broader problem. Researchers investigated different approaches to identify code clones [27]. There are a few approaches attempting to detect pure *Type-4* clones [92, 123, 124], but these techniques have been implemented to only detect C clones, which are programming language specific. Recently, researchers started to use machine learning approaches to detect clones [195, 206, 221, 228] including *Type-4* clones. Different from the scenario we proposed in this paper, clone detection uses source code only, while we also consider textual description of the changes. So we customize the clone detection approaches and applied them in a different scenario, that is identifying redundant changes in forks. As a future direction, it would be interesting to integrate and evaluate more sophisticated clone detection mechanisms as a similarity measure for the patch content clue in our approach.

6.7 Summary

In this chapter, we presented our second tooling intervention to address the inefficiencies of redundant development in fork-based development environment. This is complementary to other solutions described before (Chapter 3 and 5), and see thesis overview in Figure 1.1.

To achieve our goal, we design an approach to extract clues of similarity between code changes and train a machine learning model to predict redundant code changes as early as possible. We evaluated the effectiveness from both the maintainer's and the developer's perspectives. The result shows that we achieve 57–83% precision for detecting duplicate code changes from maintainer's perspective, and we could save developers' effort of 1.9–3.0 commits on average. Also, we show that our approach significantly outperforms existing state-of-art and provide anecdotal evidence of the usefulness of our approach from both maintainer's and developer's perspectives.

To summarize, we contribute (a) an analysis of the redundant development problem, (b) an approach to automatically identify duplicate code changes using natural language processing and machine learning, (b) clues development for indicating redundant development, beyond just title and description, (c) evidence that our approach outperforms the state-of-the-art, and (d) anecdotal evidence of the usefulness of our approach from both the maintainer's and the developer's perspectives.

Chapter 7

Future Work

This section describes two future research directions based on this thesis, including keeping on improving the collaboration efficiency for distributed software development (Section 7.1) and exploring other forms of collaboration, such as in interdisciplinary software teams (Section 7.2).

7.1 Improving coordination capability in fork-based development

The findings from the studies described previous chapters suggest research and tooling opportunities for improving fork-based development mechanism, such as assisting in finding relevant artifacts, finding potential collaborators, and diagnosing particular kinds of interactions in projects [106] as shown in Figure 7.1.

Identifying Lost Contribution in Forks. As described in Section 1, when the number of forks increases, it is hard for developers to maintain an overview in the community. This would further lead to problems like lost contribution and suboptimal forking point. In addition, the evaluation result of INFOX (see Section 5.4) confirmed that there exist potentially reusable features in other forks, but there is no way to find out based on current tool support. For example, one of our interviewees found a fork he/she has not seen before, and said: *"If it only exists in this fork, then I want to somehow get this fork into my fork."* This motivates us to identify lost contributions in forks, help developers find features of interests and enhance collaboration.

We will use the information of unmerged code changes of forks (which is the output of INFOX) as input, and identify pairs of forks that have similar/related code changes by calculating the similarity. As we have built forks-insight.com [184], and it already stored unmerged code summary of forks of many GITHUB projects. We will calculate the similarity between two forks to see if their code changes are similar (such as changing the same files, implementing the same features, testing the same configuration). Once we found a pair of forks that are potentially related, we will send out an email to inform two fork owners that there exists a fork that is potentially feature of interests.

We will evaluate the effectiveness and usefulness of our approach by asking *To what extent do developers agree with our detection result? Can our approach help developers to find the*

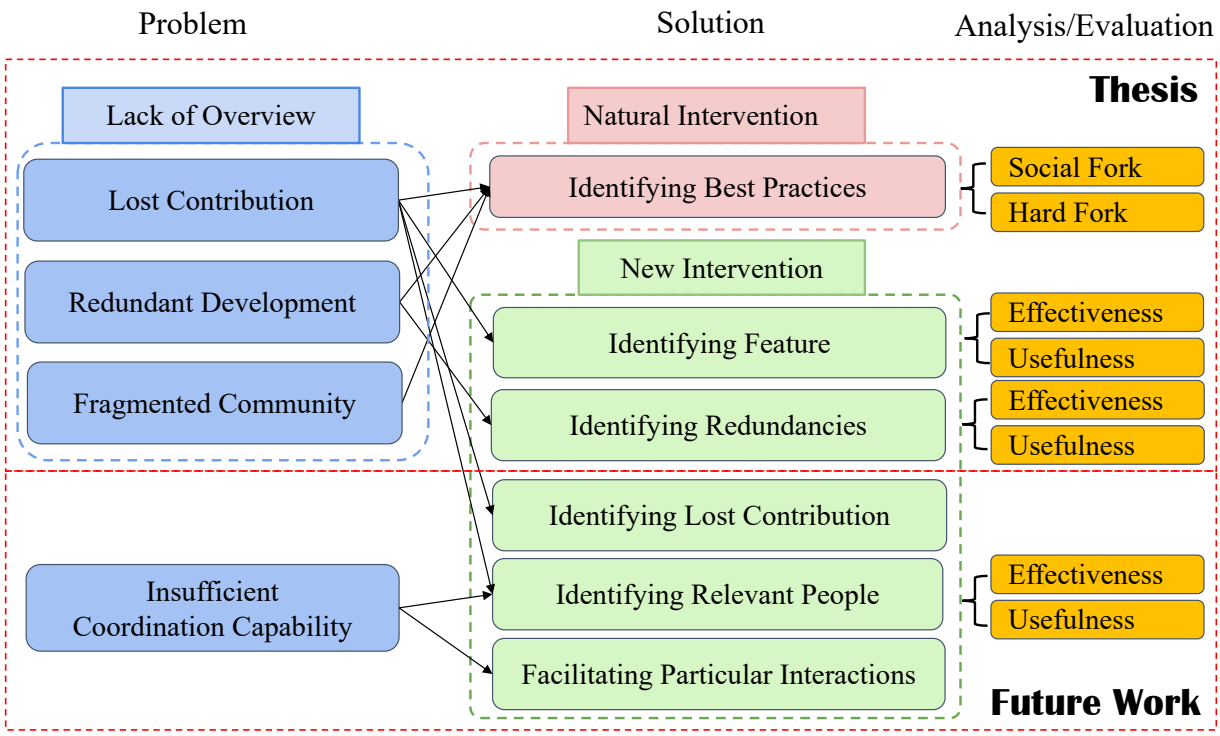


Figure 7.1: Future work: Improving coordination capability in fork-based development (Extension of Figure 1.1.

contributions they are interested in? Can our approach enhance collaboration between developers? We would like to conduct a user study. Once we detect related fork activities, we will reach out to corresponding fork owners, and present our prediction result to them, then we will track their activities to see if it does help them to collaborate. We will ask for their feedback and see if there are any actionable insights. For example, we expect to see our detection result could help developers find contributions that they have not seen before, and they are willing to reuse the contribution that we recommended.

In addition, after we identify the potentially reusable code changes that might benefit the larger community, we would design method to automatically wrap code changes into pull requests to help the community get more contribution without increasing the workload of project maintainers too much.

Identifying Relevant Collaborators. Another interesting direction is finding relevant people, rather than relevant source code, along the line of Herbsleb's work [106], but with new challenges. Researchers have designed method to locate people with the right expertise in a distributed setting. For example, Ehrlich et al. [75] conducted a social network analysis to understand how people find different kinds of expertise. Also, researchers have designed tools for finding expertise in software development projects, using data from the version control system about which developers have contributed to what parts of the code[146, 155]. Different from these existing work, there are new challenges in fork-based development and new requirements

for open-source communities. For example, open-source communities are interested in attracting more contributors, but there might be learning curve for entering the community. Therefore, it is important to assign newcomers a mentor and help them to get familiar with the project and submit high quality patches [47, 82].

Facilitating Collaborating across Fragmented Communities. As we described in Chapter 3 and 4, hard forks are fragmenting the community, and from our interviews we see collaboration opportunities among these sub-communities, which may allow new forms of collaboration across multiple hard forks and projects as part of a larger community.

For example, *Ultimaker* is a hard fork of the *Marlin* project, we observed that in a pull request for Marlin[1] for an issue that was independently fixed with a different pull request in *Ultimaker* 2 years earlier.[2] Therefore, we would like to design interventions that could build a bridge between disconnect communities, which potentially encourage the contribution within a community to benefit the larger community.

Specifically, we could design: (1) an early warning system that alerts upstream maintainers of emerging hard forks (e.g., external bots), which maintainers could use to encourage collaboration over competition and fragmentation if desired, (2) a way to declare the intention behind a fork (e.g., explicit GitHub support) and dashboard to show how multiple projects and important hard forks interrelate (e.g., pointing to hard forks that provide ports for specific operating systems), and (3) means to identify the essence of the novel contributions in forks.

7.2 Exploring Different Forms of Collaboration

The approach to research that I used in this thesis has a wider applicability than just the fork-based development. One of the future research direction could be exploring other forms of collaboration, such as in interdisciplinary software teams.

For example, the advances in machine learning (ML) have stimulated widespread interest in integrating AI capabilities into software and services [154]. Therefore the software development team is mixed with data scientists and software engineers, with the aim of increasing collaboration and communication among developers, operations professionals, and data scientists. In a ML pipeline, there are in general two phases: an *exploratory phase* and a *production phase*. Data scientists mainly work in the exploratory phase to train an off-line ML model and then deliver it to software engineers who work in the production phase to integrate the model into the production codebase. However, since data scientists are focusing on improving ML algorithms to have better prediction results often without thinking enough about the production environment, software engineers need to redo some of the exploratory work in order to integrate it into production code successfully. Additionally, once the exploratory code is integrated into production, it is non-trivial to provide feedback from the production phase to improve the experiments phase again. Furthermore, as the team is multidisciplinary, the communication time and barrier of coordination will increase [87]. Similar problems also exist in building scientific software systems:

[1] `https://github.com/MarlinFirmware/Marlin/pull/10119`
[2] `https://github.com/Ultimaker/Ultimaker2Marlin/pull/118`

In most cases, the researchers who develop the scientific software may not have the required knowledge on the best practices of software maintainability and sustainability which are needed for reproducibility of simulation results.

To facilitate the collaboration within interdisciplinary teams, we could use the software engineering principles and combine insights and theories from other disciplines, which could draw insights and methods from my previous research experiences. We believe that multidisciplinary distributed collaboration is paramount for developing modern software in the future and that we will have good chances to facilitate the development processes in a way that stakeholders can cope with successfully.

Chapter 8

Conclusion

By understanding how software developers collaborate in distributed settings using fork-based development mechanisms, we have the opportunities to assist developers in both open-source and in industry to coordinate in such environment. In this dissertation, we have used mixed-method approaches to investigate the problem space of fork-based development by identifying the collaboration inefficiencies, then we proposed, designed, and evaluated interventions that aiming for mitigating the inefficiencies. Specifically, we discussed the problem of lack of an overview in fork-based development and corresponding inefficiencies (*e.g.*, lost contribution, redundant development, and fragmented community) in Chapter 1 and how we designed measures to quantify these inefficiencies in Section 3.2. Then we described the details of the complementary solutions: First, we used cross-sectional correlational study to test the feasibility of suggesting the natural interventions that are correlated with higher collaboration efficiency to other open-source communities that have less efficiencies (Chapter 3); Second, we designed new interventions in the forms of tooling to improve current forking mechanism to identifying features in forks (Chapter 5) and detecting potentially duplicate pull requests (Chapter 6). To evaluate these interventions, we conducted both quantitative study to test the effectiveness in a large scale and test the usefulness by conducting human-subjective studies with open-source developers. During the process of researching the problem and solution, we also borrowed insights from other disciplines to better understand the problem from different perspectives and design evaluations, such as from organizational theory.

There are a few *limitations* related to the assumptions that we made in the projects. (1) An open question to the generalizability of the findings in this thesis is whether they still hold in other forking platforms, such as GITLAB and Bitbucket. In this thesis, we only studied GITHUB, which is the most popular platform that supports fork-based development. Although these platforms virtually share the same features, but they have different emphasis [1, 117]. Therefore, it is interesting to look at different and the trade-offs between platforms. As a starting point, we have studied forks that were originated on GITHUB and then migrated to GITLAB [30] by mining the Software Heritage Graph Dataset [172], which contains the forking history of both GITHUB and GITLAB, although the dataset is not reliable and complete enough. Similarly, Pietri et al. [173] has found large amount of forks that cannot be detected through GITHUB API. In the future, we would like to build our own dataset and study the differences between different platforms and see if there is any inefficiencies in the cross-platform environment and what are the new challenges.

(2) Another limitation of my work is its construct validity. In particular, the method of the approximation of measuring centralized management and project modularity is representative. We only focuses on projects that are using GITHUB issue tracker to manage planned changes, but other issue trackers has been used with GITHUB as well. Future work improving either the measurement or the indicator for project context factors would greatly further our understanding of the concept.

There are at least two future research directions based on this thesis (Chapter 7), including keeping on improving the collaboration efficiency for distributed software development and exploring other forms of collaboration, such as in interdisciplinary software teams.

Bibliography

[1] Atlassian bitbucket vs. gitlab. GitLab documentation. URL https://about.gitlab.com/devops-tools/bitbucket-vs-gitlab.html. 8

[2] A short history of git. Online Documentation. URL https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git. 1

[3] Intervention theory wikipedia. Wikipedia. URL https://en.wikipedia.org/wiki/Intervention_theory. 2

[4] Github network view. https://help.github.com/en/articles/viewing-a-repositorys-network, 2008. 4.3.3

[5] Requirement of "claiming" tickets in django project, 2011. URL https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/submitting-patches/#claiming-tickets. 3.1.2

[6] How to write the perfect pull request, 2015. URL https://github.blog/2015-01-21-how-to-write-the-perfect-pull-request/. 3.2.8

[7] Dear github issue 175: Better overview over forks, 2016. URL https://github.com/dear-github/dear-github/issues/175. 1.1, 5.1

[8] Dear github issue 191: Feature: Work in progress pull requests, 2016. URL https://github.com/dear-github/dear-github/issues/191. 3.8.2

[9] May the fork be with you: A short history of open source forks, 2016. URL https://thenewstack.io/may-fork-short-history-open-source-forks/. 2.1

[10] Wip app for github, 2016. URL https://github.com/apps/wip. 3.8.2

[11] Lovely forks browser extension: Show notable forks of github repositories under their names, 2017. URL https://github.com/musically-ut/lovely-forks. 3.8.2

[12] Github pull request triage, 2017. URL http://prs.mozilla.io/. 1.1, 3.8.2

[13] Replication package, 2019. URL https://github.com/shuiblue/ForkingEfficiencyPaper. 3.2, 3.2.3, 3.5

[14] Github wikipedia. Wikipedia, 2020. URL https://en.wikipedia.org/wiki/GitHub. 1

[15] Appendix. https://github.com/shuiblue/ICSE20-hardfork-appendix, 2020. 4.2.4, 4.3.1

[16] Muhammad Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. Mining duplicate questions in stack overflow. pages 402–412, New York, NY, USA, 2016. ACM. 6.6

[17] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Ștefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible product line engineering with a virtual platform. In *Comp. Int'l Conf. Software Engineering (ICSE)*, pages 532–535. ACM, 2014. ISBN 978-1-4503-2768-8. 3.8.2, 4.3.3, 5.6

[18] Amirhosein Emerson Azarbakht. *Longitudinal Analysis of Collaboration in Forked Open Source Software Development Projects*. PhD thesis, Oregon State University, 2017. 1.1, 3.1.3

[19] Carliss Y Baldwin and Kim B Clark. The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Management Science*, 52(7):1116–1127, 2006. 3.1.1, 3.1.2

[20] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000. 3.7.1

[21] Sergio Bandinelli, Elisabetta Di Nitto, and Alfonso Fuggetta. Supporting cooperation in the spade-1 environment. *IEEE Trans. Softw. Eng. (TSE)*, 22(12):841–865, 1996. 2.2

[22] Jakob E Bardram and Thomas R Hansen. Context-based workplace awareness. *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 19(2):105–138, 2010. 2.3

[23] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. Int'l Conf. Software Engineering (ICSE)*, volume 1, pages 134–144. IEEE, 2015. 5.1, 5.2, 5.2.1, 5.2.2, 5.4, 5.5

[24] Earl T. Barr, Christian Bird, Peter C. Rigby, Abram Hindle, Daniel M. German, and Premkumar Devanbu. Cohesive and isolated development with branches. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, pages 316–331, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 1

[25] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 354–364. ACM, 2011. 3.2.5

[26] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 12–23. ACM, 2014. 3.1.4

[27] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng. (TSE)*, 33(9), 2007. 6.3.2, 6.6

[28] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. Three cases of feature-based variability modeling in industry. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*, pages 302–319. Springer, 2014. 1.1

[29] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *Proc. Int'l Conf. Software Maintenance and Evolution(ICSME)*, pages 337–345. IEEE, 2008. 6.6

[30] Avijit Bhattacharjee, Sristy Sumana Nath, Shurui Zhou, Debasish Chakroborti, Banani Roy, Chanchal Roy, and Kevin Schneider. An exploratory study to find motives behind cross-platform forks from software heritage dataset. 03 2020. 8

[31] Marco Biazzini and Benoit Baudry. 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*, pages 37–43. ACM, 2014. 2.1, 3.2.1

[32] Jacob T Biehl, Mary Czerwinski, Greg Smith, and George G Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. pages 1313–1322. ACM, 2007. 2.3

[33] James M Bieman and Linda M Ott. Measuring functional cohesion. *IEEE Trans. Softw. Eng. (TSE)*, 20(8):644–657, 1994. 3.2.5

[34] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in oss projects. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 26–26. IEEE, 2007. 2.1

[35] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. pages 6–6. IEEE, 2007. 2.1

[36] Christian Bird, Thomas Zimmermann, and Tom Zimmermann. Assessing the value of branches with what-if analysis. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. Association for Computing Machinery, Inc., November 2012. 1, 5.5

[37] Jürgen Bitzer and Philipp JH Schröder. The impact of entry and competition by open source software on innovation activity. *The economics of open source software development*, pages 219–245, 2006. 1

[38] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2016. 2.1, 3.1.3, 4.3.3

[39] Jan Bosch. From software product lines to software ecosystems. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 111–119. Carnegie Mellon University, 2009. 5.6

[40] E. Bouwers, A. van Deursen, and J. Visser. Evaluating usefulness of software metrics: An industrial experience report. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 921–930, May 2013. 3, 3.8.2

[41] Jordi Brandts, David J Cooper, et al. Truth be told an experimental study of communication and centralization. Technical report, 2018. 3.1.2

[42] Pete Bratach. Why do open source projects fork? Blog Post, 2017. URL `https://thenewstack.io/open-source-projects-fork/`. 2.1, 4.1

[43] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011. 6.6

[44] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Improving the tokenisation of identifier names. *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 130–154, 2011. 5.2.3

[45] Yuangfang Cai and Sunny Huynh. An evolution model for software modularity assessment. pages 3–3. IEEE, 2007. 3.2.5

[46] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013. ISBN 1617290955, 9781617290954. 3.8.2

[47] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. Who is going to mentor newcomers in open source projects? In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 44:1–44:11. ACM, 2012. ISBN 978-1-4503-1614-9. 3.8.2, 7.1

[48] Marcelo Cataldo, Patrick A Wagstrom, James D Herbsleb, and Kathleen M Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 353–362, 2006. 1.2, 2.3

[49] I. Chawla and S. K. Singh. Performance evaluation of vsm and lsi models to determine bug reports similarity. In *2013 Sixth International Conference on Contemporary Computing (IC3)*, pages 375–380, 2013. 6.4.1, 6.4.1

[50] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *Proc. Int'l Workshop on Program Comprehension (IWPC)*, pages 241–247. IEEE, 2000. 5.5

[51] Bee Bee Chua. A survey paper on open source forking motivation reasons and challenges. 2017. 2.1, 4.1

[52] Bredan Cleary and Chris Exton. *Assisting Concept Location in Software Comprehension*. PhD thesis, University of Limerick, 2007. 5.4.1

[53] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, 2009. 5.4.1, 5.5

[54] Michael L Collard, Michael John Decker, and Jonathan I Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 516–519. IEEE, 2013. 5.3

[55] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968. 3.1.1, 3.7.1

[56] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Softw. Eng. (TSE)*, 35(5):684–702, 2009. 5.5

[57] John W Creswell and J David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017. 4.2

[58] Davor Čubranić and Gail C Murphy. Hipikat: Recommending pertinent software devel-

opment artifacts. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 408–418. IEEE Computer Society, 2003. 5.4.1

[59] Davor ČubraniĆ, Gail C Murphy, Janice Singer, and Kellogg S Booth. Learning from project history: a case study for software development. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 82–91. ACM, 2004. 5.4.1

[60] Davor Cubranic, Gail C Murphy, Janice Singer, and Kellogg S Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng. (TSE)*, 31(6):446–465, 2005. 5.4.1

[61] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 1277–1286. ACM, 2012. 1, 1.1, 1.1, 2.1, 2.3, 3.7.2, 4.3.3, 4.3.3, 5.1

[62] Laura Dabbish, Colleen Stuart, Jason Tsay, and James Herbsleb. Leveraging transparency. *IEEE Software*, 30(1):37–43, 2013. 1.1, 2.1, 4.3.3

[63] Daniela Damian, Luis Izquierdo, Janice Singer, and Irwin Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proc. Int'l Conf. Global Software Engineering*, pages 81–90. IEEE, 2007. 1.2, 2.3

[64] Paul B. de Laat. Governance of open source software: state of the art. *Journal of Management & Governance*, 11(2):165–177, May 2007. 1

[65] Cleidson RB De Souza, David Redmiles, and Paul Dourish. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work*, pages 105–114. ACM, 2003. 2.3

[66] Premkumar Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 108–119. IEEE, 2016. 3.1.4

[67] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, pages 341–350. IEEE, 2015. 5.5

[68] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25 (1):53–95, 2013. 5.4.1, 5.5

[69] James Dixon. Forking protocol: Why, when, and how to fork an open source project. Blog Post, 2009. URL `https://jamesdixon.wordpress.com/2009/05/13/different-kinds-of-open-source-forks-salad-dinner-and-fish/`. 2.1, 4.1

[70] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work*, Proc. Conf. Computer Supported Cooperative Work (CSCW), pages 107–114. ACM, 1992. ISBN 0-89791-542-9. 1, 1.2, 2.3

[71] Yael Dubinsky, Julia Rubin, Theodore Berger, Slawomir Duszynski, Matthias Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 25–34. IEEE, 2013. 1, 1.1, 3.8.2, 5.5, 5.7, 6.1

[72] Anh Nguyen Duc, Audris Mockus, Randy Hackbarth, and John Palframan. Forking and coordination in multi-platform development: A case study. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*, pages 59:1–59:10. ACM, 2014. 1.1

[73] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 14(4): 323–368, 2005. 2.3

[74] Marc Eaddy, Alfred V Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 53–62. Ieee, 2008. 5.4.1

[75] Kate Ehrlich and Klarissa Chang. Leveraging expertise in global software teams: Going outside boundaries. In *Proc. Int'l Conf. Global Software Engineering*, pages 149–158. IEEE, 2006. 7.1

[76] SC Eick, Joseph L Steffen, and Eric E Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng. (TSE)*, 18(11):957–968, 1992. 2.3

[77] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Softw. Eng. (TSE)*, 29(3):210–224, 2003. 5.5

[78] Jason B Ellis, Shahtab Wahid, Catalina Danis, and Wendy A Kellogg. Task and social visualization in software development: evaluation of a prototype. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 577–586. ACM, 2007. 2.3

[79] Thomas J Emerson. A discriminant metric for module cohesion. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 294–303. IEEE Press, 1984. 3.2.5, 5.2.1

[80] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng. (TSE)*, pages 1146–1170, 2002. 5.4.1

[81] Neil A Ernst, Steve Easterbrook, and John Mylopoulos. Code forking in open-source software: a requirements perspective. *arXiv preprint arXiv:1004.2889*, 2010. 1, 2.1, 4.1

[82] Fabian Fagerholm, Alejandro S Guinea, Jürgen Münch, and Jay Borenstein. 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)*, page 55. ACM, 2014. 3.8.2, 7.1

[83] Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachselt. Featurecommander: Colorful #ifdef world. In *Proc. Int'l Software Product Line Conf. (SPLC)*, page 48. ACM, 2011. 5.3

[84] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Proc.*

Int'l Conf. Software Maintenance (ICSM), pages 391–400. IEEE, 2014. 3.8.2, 4.3.1, 4.3.3

[85] Karl Fogel. *Producing open source software: How to run a successful free software project*. " O'Reilly Media, Inc.", 2005. 2.1, 2.1, 4, 4.1

[86] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010. 5.2.2, 5.2.2

[87] Frederick P Brooks. The mythical man-month. *Datamation*, 20(12):44–52, 1974. 7.2

[88] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, pages 119–139, 1997. 6.4.2

[89] Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 387–396. IEEE Computer Society, 2004. 2.3

[90] Randall Frost. Jazz and the eclipse way of collaboration. *IEEE software*, 24(6), 2007. 2.3

[91] Kam Hay Fung, Aybüke Aurum, and David Tang. Social forking in open source software: An empirical study. In *CAiSE Forum*, pages 50–57. Citeseer, 2012. 1.1

[92] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2008. 6.6

[93] Jonas Gamalielsson and Björn Lundell. Sustainability of open source software communities beyond a fork: How and why has the libreoffice project evolved? *Journal of Systems and Software*, 89:128–145, 2014. 2.1

[94] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 351–360. IEEE, 2009. 5.5

[95] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002. 5.1, 5.2.2, 5.6

[96] Georgios Gousios. The ghtorent dataset and tool suite. pages 233–236. IEEE Press, 2013. 1, 1.1, 3.3, 4.2.2, 6.3.2

[97] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 345–355. ACM, 2014. 1, 1.1, 1.1, 2.1, 2.1, 3, 3.2.2, 3.2.9, 3.4, 6.1, 6.5.1

[98] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. pages 384–387. ACM, 2014. 4.1, 5.4.2

[99] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. Technical report, Delft University of Technology, Software Engineering Research Group, 2014. 1.1, 2.1

[100] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 285–296. IEEE, 2016. 1.1, 1.1, 6.2, 5

[101] Aaron Greenhouse and William L Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 453–463. IEEE, 2002. 2.2

[102] Carl Gutwin and Saul Greenberg. The importance of awareness for team cognition in distributed collaboration. 2001. 2.3

[103] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81. ACM, 2004. 1, 1.2, 2.3

[104] Rajesh Hegde and Prasun Dewan. Connecting programming environments to support adhoc collaboration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 178–187. IEEE Computer Society, 2008. 2.3

[105] James Herbsleb and Jeff Roberts. Collaboration in software engineering projects: A theory of coordination. *ICIS 2006 Proceedings*, page 38, 2006. 2.2

[106] James D Herbsleb. Global software engineering: The future of socio-technical coordination. In *Proc. Int'l Symposium Future of Software Engineering (FOSE)*, pages 188–198. IEEE, 2007. 1, 7.1, 7.1

[107] James D Herbsleb and Rebecca E Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 85–95. ACM, 1999. 1, 3.1.1

[108] James D. Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Trans. Softw. Eng. (TSE)*, 29(6):481–494, 2003. 2.3

[109] James D Herbsleb, Audris Mockus, Thomas A Finholt, and Rebecca E Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 319–328. ACM, 2000. 2.3

[110] Kim Herzig and Andreas Zeller. Untangling changes. *Unpublished manuscript, September*, 37:38–40, 2011. 5.5

[111] Kim Herzig and Andreas Zeller. The impact of tangled code changes. pages 121–130. IEEE Press, 2013. ISBN 978-1-4673-2936-1. 5.5

[112] Kim Herzig and Andreas Zeller. The impact of tangled code changes. pages 121–130. IEEE Press, 2013. ISBN 978-1-4673-2936-1. 5.1

[113] Kim Herzig and Andreas Zeller. The impact of tangled code changes. pages 121–130. IEEE, 2013. 5.1

[114] Lyndon Hiew. Assisted detection of duplicate bug reports. 2006. 6.6

[115] Emily Hill, Lori Pollock, and K Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 14–23. ACM, 2007. 5.4.1

[116] Emily Hill, Lori Pollock, and K Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proc. Int'l Conf. Software*

Engineering (ICSE), pages 232–242. IEEE, 2009. 5.4.1, 5.5

[117] Karl Hughes. Github vs. bitbucket vs. gitlab - help me decide. Blog Post, 2018. URL `https://stackshare.io/stackups/bitbucket-vs-github-vs-gitlab`. 8

[118] Riitta Jääskeläinen. Think-aloud protocol. *Handbook of translation studies*, 1:371–374, 2010. 5.4.2

[119] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 52–61, June 2008. 6.6

[120] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 52–61, June 2008. 6.3.2

[121] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013. 6.4.2

[122] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining feature traceability with embedded annotations. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 61–70. ACM, 2015. 4.3.1

[123] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pages 81–92. ACM, 2009. 6.6

[124] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 96–105. IEEE Computer Society, 2007. 6.6

[125] Natalia Juristo and Omar S Gómez. Replication of software engineering experiments. In *Empirical software engineering and verification*, pages 60–88. Springer, 2010. 4.1, 4.2

[126] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M German. Open source-style collaborative development practices in commercial projects using github. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 574–585. IEEE Press, 2015. 3.5

[127] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 351–360. ACM, 2011. 5.1

[128] Bruce Kogut and Anca Metiu. Open-source software development and distributed innovation. *Oxford review of economic policy*, 17(2):248–264, 2001. 3.1.1

[129] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology (IST)*, 49(3):230–243, 2007. 5.5

[130] Thomas K. Landauer and Susan Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge. *Psychological Review*, 104(2):211–240, 1997. 6.4.1

[131] Andrew M St Laurent. *Understanding open source and free software licensing: guide to*

navigating licensing issues in existing & new software. " O'Reilly Media, Inc.", 2004. 2.1, 2.1, 4, 4.1

[132] Paul Layzell, O Pearl Brereton, and Andrew French. Supporting collaboration in distributed software engineering teams. In *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*, pages 38–45. IEEE, 2000. 2.2

[133] Sungjick Lee and Han-joon Kim. News keyword extraction for topic tracking. pages 554–559. IEEE, 2008. 5.2.3

[134] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Precise semantic history slicing through dynamic delta refinement. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 495–506, Septermber 2016. 5.5

[135] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Trans. Softw. Eng. (TSE)*, 44(2):182–201, 2017. 4.3.4, 5.1, 5.5

[136] Zhixing Li, Gang Yin, Yue Yu, Tao Wang, and Huaimin Wang. Detecting duplicate pull-requests in github. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, page 20. ACM, 2017. 3.7.3, 3.8.2, 6.3.2, 6.5.2, 6.6

[137] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2010. 5.4.1, 5.4.1

[138] Max Lillack, Ştefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. Intention-based integration of software variants. In *Proc. Int'l Conf. Software Engineering (ICSE)*, ICSE '19, pages 831–842, Piscataway, NJ, USA, 2019. IEEE Press. 4.3.1

[139] Bin Lin, Gregorio Robles, and Alexander Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *Proc. Int'l Conf. Global Software Engineering*, pages 66–75. IEEE, 2017. 1

[140] Alec Liu. Who's building bitcoin? an inside look at bitcoin's open source development. *Motherboard*, 2013. 3.1.2

[141] K. Liu, H. Beng Kuan Tan, and H. Zhang. Has this bug been reported? In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 82–91, Oct 2013. 6.6

[142] K. Liu, H. Beng Kuan Tan, and H. Zhang. Has this bug been reported? In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 82–91, Oct 2013. 6.3.2

[143] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006. 3.1.1

[144] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I Maletic. An information retrieval approach to concept location in source code. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 214–223. IEEE, 2004. 5.5

[145] Steve McConnell. Rapid development (vol. 1556159005), 1996. 2.2

[146] David W McDonald and Mark S Ackerman. Expertise recommender: a flexible recom-

mendation system and architecture. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 231–240, 2000. 7.1

[147] Nora McDonald and Sean Goggins. Performance and participation in open source software on github. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, page 139–144, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319522. 1.1

[148] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 495–518. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. 5.4.1

[149] Ines Mergel. Open collaboration in the public sector: The case of social coding on github. *Government Information Quarterly*, 32(4):464–472, 2015. 2.1

[150] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988. 3.7.1

[151] Peiwei Mi, Walt Scacchi, et al. Modeling articulation work in software engineering processes. In *Proceedings. First International Conference on the Software Process,*, pages 188–201. IEEE, 1991. 2.3

[152] Vishal Midha and Prashant Palvia. Factors affecting the success of open source software. *J. Syst. Softw.*, 85(4):895–905, April 2012. ISSN 0164-1212. 3.1.1

[153] Tommi Mikkonen and Linus Nyman. To Fork or Not to Fork: Fork Motivations in SourceForge Projects. *Int. J. Open Source Softw. Process.*, 3(3):1–9, July 2011. ISSN 1942-3926. 5.5, 5.7

[154] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 96–107. ACM, 2016. 7.2

[155] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 503–512. IEEE, 2002. 7.1

[156] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 11(3):309–346, 2002. 2.1, 2.3

[157] Leticia Montalvillo and Oscar Díaz. Tuning GitHub for spl development: branching models & repository operations for product engineers. In *Proceedings of the 19th International Conference on Software Product Line*, pages 111–120. ACM, 2015. 4.3.3

[158] Gail Cecile Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, 1996. 5.5

[159] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 287–297. IEEE Computer Society, 2009. 5.1

[160] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yun-

wen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the international workshop on Principles of software evolution*, pages 76–85. ACM, 2002. 3.1.3

[161] Linus Nyman. Hackers on forking. In *Proceedings of The International Symposium on Open Collaboration*, page 6. ACM, 2014. 2.1, 2.1, 4, 4.1

[162] Linus Nyman and Tommi Mikkonen. To fork or not to fork: Fork motivations in sourceforge projects. In *IFIP International Conference on Open Source Systems*, pages 259–268. Springer, 2011. 2.1, 4.1, 4.2.2

[163] Linus Nyman, Tommi Mikkonen, Juho Lindman, and Martin Fougère. Perspectives on code forking and sustainability in open source software. *Why Linux on't fork*, 1999. 2.1

[164] Linus Nyman, Tommi Mikkonen, Juho Lindman, and Martin Fougère. Perspectives on code forking and sustainability in open source software. *Open Source Systems: Long-Term Sustainability*, pages 274–279, 2012. 2.1

[165] Gary M Olson and Judith S Olson. Distance matters. *Human–computer interaction*, 15 (2-3):139–178, 2000. 1

[166] Inah Omoronyia, John Ferguson, Marc Roper, and Murray Wood. Using developer activity data to enhance awareness during collaborative software development. *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 18(5-6):509, 2009. 2.3

[167] Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 6813, pages 155–178. Springer, 2011. 3.1.3

[168] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. Needfeed: Taming change notifications by modeling code relevance. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 665–676. ACM, 2014. 4.3.3, 4.3.3

[169] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. 3.1.1, 3.7.1

[170] Maksym Petrenko, Václav Rajlich, and Radu Vanciu. Partial domain comprehension in software evolution and maintenance. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 13–22. IEEE, 2008. 5.5

[171] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121. IEEE Press, 2013. 2.3

[172] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The Software Heritage Graph Dataset: Large-scale analysis of public software development history. In *MSR 2020: The 17th International Conference on Mining Software Repositories*. IEEE, 2020. 8

[173] Antoine Pietri, Stefano Zacchiroli, and Guillaume Rousseau. Forking without clicking: on how to identify software repository forks. 2020. 8

[174] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proc. Int'l*

Conf. Software Engineering (ICSE), pages 465–475. IEEE, 2003. 6.3.2, 6.6

[175] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 37–48. IEEE, 2007. 5.5

[176] Phanish Puranam, Oliver Alexy, and Markus Reitzig. What's "new" about new forms of organizing? *Academy of Management Review*, 39(2):162–180, 2014. 3.1.2

[177] Md Masudur Rahman, Saikat Chakraborty, Gail E. Kaiser, and Baishakhi Ray. A case study on the impact of similarity measure on information retrieval based software engineering tasks. *CoRR*, abs/1808.02911, 2018. 6.4.1

[178] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 271–278, 2002. 5.5

[179] Ayushi Rastogi and Nachiappan Nagappan. 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)*, volume 1, pages 102–111. IEEE, 2016. 1, 1.1, 2.1, 2.1, 3

[180] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 367–377. IEEE, 2013. 2.1, 4.3.1, 4.3.3

[181] Eric S Raymond. *The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary*. " O'Reilly Media, Inc.", 2001. 2.1, 2.1, 4, 4.1

[182] L. Ren, S. Zhou, and C. Kästner. Poster: Forks insight: Providing an overview of github forks. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 179–180, 2018. 5

[183] Luyao Ren. Automated patch porting across forked projects. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 1199–1201, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5572-8. 4.3.3

[184] Luyao Ren, Shurui Zhou, and Christian Kästner. Poster: Forks insight: Providing an overview of github forks. In *Proceedings of the Companion of the International Conference on Software Engineering (ICSE)*, New York, NY, 2018. ACM Press. Poster. 1.1, 3.8.2, 7.1

[185] Luyao Ren, Shurui Zhou, Christian Kästner, and Andrzej Wąsowski. Identifying redundancies in fork-based development. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, 2019. 1.2, 3.7.3, 3.8.2, 6

[186] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 14–23. IEEE, 2010. 5.4.1

[187] Peter C Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 541–550. ACM, 2011. 2.3

[188] Martin P Robillard. Automatic generation of suggestions for program investigation. In

SIGSOFT Softw. Eng. Notes, volume 30, pages 11–20. ACM, 2005. 5.5

[189] Martin P Robillard, David Shepherd, Emily Hill, K Vijay-Shanker, and Lori Pollock. An empirical study of the concept assignment problem. *School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2007.3*, 2007. 5.4.1

[190] Gregorio Robles and Jesús M. González-Barahona. A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes. In *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*, pages 1–14, 2012. doi: 10.1007/978-3-642-33442-9_1. 2.1, 4.1, 4.2.2, 5.5, 5.7

[191] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009. 6.3.2, 6.6

[192] Julia Rubin and Marsha Chechik. A framework for managing cloned product variants. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1233–1236. IEEE Press, 2013. 4.3.3, 5.6

[193] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 499–510, May 2007. 6.6

[194] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Commun. ACM*, pages 58–66, 2018. 6.1

[195] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 354–365, New York, NY, USA, 2018. ACM. 6.6

[196] Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015. 4.2.4, 5.4.2

[197] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988. 5.1, 5.2.3, 6.4.1

[198] Raghvinder Sangwan, Matthew Bass, Neel Mullick, Daniel J Paulish, and Juergen Kazmeier. *Global software development handbook*. Auerbach Publications, 2006. 2.3

[199] Anand Mani Sankar. Node.js vs io.js: Why the fork?!? Blog Post, 2015. URL `http://anandmanisankar.com/posts/nodejs-iojs-why-the-fork/`. 2.1

[200] Anita Sarma and Andre van der Hoek. Towards awareness in the large. In *Proc. Int'l Conf. Global Software Engineering*, pages 127–131, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2663-2. 1.2, 2.3

[201] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of the 31st International Conference on Software Engineering*, pages 23–33. IEEE Computer Society, 2009. 2.3

[202] Anita Sarma, David F. Redmiles, and André van der Hoek. Palantír: Early detection of development conflicts arising from parallel code changes. *IEEE Trans. Softw. Eng. (TSE)*,

38(4):889–908, 2012. 2.3, 6.6

[203] Stefan Schmidt. Shall we really do it again? the powerful concept of replication is neglected in the social sciences. *Review of General Psychology*, 13(2):90–100, 2009. 4.1, 4.2

[204] Bikram Sengupta, Satish Chandra, and Vibha Sinha. A research agenda for distributed software development. In *Proceedings of the 28th international conference on Software engineering*, pages 731–740. ACM, 2006. 2.3

[205] Maha Shaikh and Ola Henfridsson. Governing open source software through coordination processes. *Information and Organization*, 27(2):116–135, 2017. 3.1.2

[206] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on*, pages 1024–1028. IEEE, 2016. 6.6

[207] David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 212–224. ACM, 2007. 5.5

[208] Yoonki Song, Xiaoyin Wang, Tao Xie, Lu Zhang, and Hong Mei. Jdf: detecting duplicate bug reports in jazz. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 315–316. ACM, 2010. 6.3.2, 6.6

[209] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009. 4.2.3

[210] Kathy Spurr, Paul Layzell, Leslie Jennison, and Neil Richards. *Computer support for co-operative work*. John Wiley & Sons, Inc., 1994. 2.2

[211] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. Concepts, operations, and feasibility of a projection-based variation control system. In *Proc. Int'l Conf. Software Maintenance and Evolution(ICSME)*, pages 323–333. IEEE, 2016. 1.1, 3.8.2, 4.3.3

[212] Igor Steinmacher, Gustavo H. L. Pinto, Igor Scaliante Wiese, and Marco Aurélio Gerosa. Almost there: A study on quasi-contributors in open-source software projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1–12, 2018. 1.1, 3.1.3, 6.1

[213] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Shared waypoints and social tagging to support collaboration in software development. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 195–198. ACM, 2006. 5.5

[214] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *31st International Conference on Software Maintenance and Evolution*, Proc. Int'l Conf. Software Maintenance and Evolution(ICSME), 2015. 1.1, 2.1, 5.4.1, 5.4.1, 5.5, 5.7, 6.1

[215] Ching Y Suen. N-gram statistics for natural language understanding and text processing. *IEEE transactions on pattern analysis and machine intelligence*, (2):164–172, 1979. 5.2.3

[216] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate

retrieval of duplicate bug reports. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 253–262. IEEE, 2011. 6.6

[217] Marcel Taeumel, Stephanie Platz, Bastian Steinert, Robert Hirschfeld, and Hidehiko Masuhara. Unravel programming sessions with thresher: Identifying coherent and complete sets of fine-granular source code changes. *Information and Media Technologies*, 12:24–39, 2017. 5.5

[218] Lei Tang and Huan Liu. Community detection and mining in social media. *Synthesis Lectures on Data Mining and Knowledge Discovery*, pages 1–137, 2010. 5.2.2, 5.4.1

[219] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119, Los Alamitos, CA, 1999. IEEE Computer Society. ISBN 1-58113-074-0. 3.1.3

[220] Stephanie D Teasley, Lisa A Covi, Mayuram S. Krishnan, and Judith S Olson. Rapid software development through team collocation. *IEEE Trans. Softw. Eng. (TSE)*, 28(7): 671–683, 2002. 2.2

[221] R. Tekchandani, R. K. Bhatia, and M. Singh. Semantic code clone detection using parse trees and grammar recovery. In *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, pages 41–46, 2013. 6.6

[222] Linus Torvalds. The linux edge. *Communications of the ACM*, 42(4):38–38, 1999. 3.1.1

[223] Linux Torvalds. Initial revision of git. Commit, 2005. URL `https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290`. 1

[224] Christoph Treude and Margaret-Anne Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 365–374. IEEE, 2010. 1.2, 2.3

[225] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 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, pages 511–522. ACM, 2018. ISBN 978-1-4503-5638-1. 3.2.9, 3.8.2

[226] Jason Tsay. *Software Developers Using Signals in Transparent Environments*. PhD thesis, Carnegie Mellon University, 4 2017. 1.1, 2.3

[227] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th international conference on Software engineering*, pages 356–366. ACM, 2014. 2.1, 3.2.9, 3.4

[228] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. pages 542–553, 2018. 6.6

[229] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. The sky is not the limit: Multitasking on GitHub projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 994–1005. ACM, 2016.

6.5.2

[230] Greg R Vetter. Open source licensing and scattering opportunism in software standards. *BCL Rev.*, 48:225, 2007. 1

[231] Adriano Vieira. I'd like to see all forked projects of one project. `https://gitlab.com/gitlab-org/gitlab-foss/issues/2406`, 2016. 1.1

[232] Robert Viseur. Forks impacts and motivations in free and open source projects. *International Journal of Advanced Computer Science and Applications*, 3(2):117–122, 2012. 2.1, 4.1

[233] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 461–470. ACM, 2008. 6.3.2, 6.6

[234] Steve Weber. *The success of open source*. Harvard University Press, 2004. 2.1

[235] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! pages 67–76, 2008. 2.1

[236] Mairieli Wessel, Bruno Mendes De Souza, Igor Steinmacher Teinmacher, Igor S Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A Gerosa. The power of bots: Understanding bots in oss projects. 2018. 6.2

[237] David A. Wheeler. Why open source software/free software (oss/fs, floss, or foss)? look at the numbers! Blog Post, 2015. URL `https://dwheeler.com/oss_fs_why.html`. 2.1, 4.1

[238] Jim Whitehead. Collaboration in software engineering: A roadmap. In *Proc. Int'l Symposium Future of Software Engineering (FOSE)*, pages 214–225. IEEE, 2007. 2.2

[239] Norman Wilde and Michael C Scully. Software reconnaissance: Mapping program features to code. *Journal of Software: Evolution and Process*, 7(1):49–62, 1995. 5.5

[240] Owen Willams. Node.js and io.js are settling their differences, merging back together. Blog Post, 2015. URL `https://thenextweb.com/dd/2015/06/16/node-js-and-io-js-are-settling-their-differences-merging-back-together` 2.1

[241] Zhang Xin, Chen Yang, Gu Yongfeng, Zou Weiqin, Xie Xiaoyuan, Jia Xiangyang, and Xuan Jifeng. How do multiple pull requests change the same code: A study of competing pull requests in Github. In *Proc. Int'l Conf. Software Maintenance and Evolution(ICSME)*, page 12, 2018. 6.6

[242] Andrew Y Yao. Cvssearch: Searching through source code using cvs comments. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, page 364. IEEE Computer Society, 2001. 5.4.1

[243] Christopher S Yoo. Open source, modular platforms, and the challenge of fragmentation. *Criterion J. on Innovation*, 1:619, 2016. 2.1

[244] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *Mining software repositories (MSR), 2015 IEEE/ACM 12th working conference on*, pages 367–371. IEEE,

2015. 2.1

[245] Yue Yu, Li Zhixing, Yin Gang, Tao Wang, and Wang Huaimin. A dataset of duplicate pull-requests in github. page 12, 2018. 3.2.3, 3.8.2, 6.1, 6.5.1, 6.6

[246] Jie Zhang, Xiaoyin Wang, Dan Hao, Bing Xie, Lu Zhang, and Hong Mei. A survey on bug-report analysis. *Science China Information Sciences*, page 1–24. 6.6

[247] Jian Zhou and Hongyu Zhang. Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 852–861, New York, NY, USA, 2012. ACM. 6.6

[248] Jian Zhou and Hongyu Zhang. Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 852–861, New York, NY, USA, 2012. ACM. 6.3.2

[249] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. Identifying features in forks. In *Proc. Int'l Conf. Software Engineering (ICSE)*, New York, NY, 5 2018. ACM Press. 1.2c, 1.1, 1.2, 3.8.2, 4.3.3, 4.3.3, 4.3.4, 5, 5.3, 6.1, 5

[250] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. What the fork: A study of inefficient and efficient forking practices in social coding. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 350–361, New York, NY, 8 2019. ACM Press. 1.3, 3, 4.2.1, 4.2.2

[251] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. How has forking changed in the last 20 years? a study of hard forks on github. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, New York, NY, 5 2020. ACM Press. 4, 4.2.2

[252] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). pages 73–83. IEEE, 2003. 3.2.5