How Developer Coreness Influences the Patch-Review Process: A Mixed-Method Study

Christian Hechtl[1*], Thomas Bock[2†], Ralf Ramsauer[3], Wolfgang Mauerer[3,4], Sven Apel[1]

[1]Saarland University, Saarbrücken, Germany.
[2]Carnegie Mellon University, Pittsburgh, USA.
[3]Technical University of Applied Sciences Regensburg, Regensburg, Germany.
[4]Siemens AG, Foundational Technologies, Germany.

[†]The majority of this work was done while Thomas Bock was with Saarland University, Germany.

Abstract

The code integration process is critical for any distributed, large-scale open-source software (OSS) project. It serves as an implicit or explicit quality control gate and is inherently of a socio-technical nature in that the bare technical act of merging new code contributions is preceded by (oftentimes engaged) discussions and reviews. Given the reasonable and widely accepted assumption that professional experience and seniority lead to higher social credit in communities, more experienced developers are expected to get favored in this process, manifesting in higher probabilities of receiving feedback on contributions, or getting contributions accepted. We conjecture that exceptions to this pattern may indicate procedural issues and examine this hypothesis through a mixed-method study. To this end, we use *developer coreness*, a continuous proxy measure of experience that measures how important and connected a developer is within a project. We then study code integration processes of 16 popular OSS projects, employing a new methodology to measure the impact of developer coreness on these processes. This allows us to identify process-deviant projects, which we investigate qualitatively to determine whether unexpected observations indicate underlying procedural issues. Our findings show that developers with higher coreness values have a higher probability of getting code contributions accepted and, in many cases, of receiving feedback. Notably, projects identified as process-deviant often exhibit signs of procedural deficiencies, highlighting the practical utility of our methodological framework.

Keywords: open-source software development, survival analysis, developer coreness

1 Introduction

Software development is a socio-technical activity, often performed by large, geographically distributed teams of developers with strongly varying background and experience. In contrast to commercial, closed-source software projects, developers in open-source software (OSS) projects are often volunteering and have the freedom of choice to join and leave projects (Lakhani and Wolf 2005). This leads to unfavorable situations when long-term developers leave a project or many inexperienced developers join a project (Foucault et al. 2015; Lin et al. 2017; Avelino et al. 2019; Schilling 2014). Transferring (implicit) knowledge across the contributors of an OSS project is known to be an effortful task (Foundjem et al. 2021).

OSS projects, their development processes, and the communities and organizations behind have received considerable attention by the research community (Foundjem et al. 2021; Bosu et al. 2017; Jiang et al. 2013; Qiu et al. 2019; Mauerer and Jaeger 2013; Gharehyazie et al. 2015). Many aspects of the development and collaboration process of OSS projects can be measured (Joblin et al. 2017a; Diamantopoulos et al. 2020; Ramsauer et al. 2019; Storey et al. 2017; Tao et al. 2014; Joblin et al. 2023; Mauerer et al. 2022), yet we lack an understanding of how a developer's position in the organizational hierarchy affects collaboration aspects such as the time it takes to get feedback on a code contribution or the probability of the acceptance of code contributions. It has been conjectured in the literature that successful collaboration interactions reduce developer turnover (as developers that have a good experience are enticed to stay longer with a project), which is, however, hard to quantify in practice and subject of an ongoing debate (Bosu and Carver 2014; Jiang et al. 2013; Bosu et al. 2017).

Specifically, our aim is to understand how the position of a developer in the organizational hierarchy influences the *feedback* on and *acceptance* of a code contribution. We aim at investigating whether a higher standing gives developers an overall better success rate in the patch-review process of OSS projects. Beside theoretical insights, analyzing the feedback and acceptance mechanisms of OSS projects provides freelancers and companies an overview of the inner mechanics of a project, which comprises valuable knowledge for successful contributions. Since especially companies are interested in fast and uncomplicated contributions to OSS projects (Butler et al. 2021), our analysis can help them to understand *how* to approach a contribution, for example, by collaborating with an experienced developer of the project.

In general, it is hard to uncover the often implicit and emerging organizational hierarchy of an OSS project (Joblin et al. 2023). As a consequence, certain aspects thereof, such as the importance of developers, have been measured in different ways in the literature (Bock et al. 2023; Joblin et al. 2023). One conjecture is that developers at the top of the organizational hierarchy (i.e., core contributors) have more success with the code contribution process in OSS projects than lower ranked developers, since

core contributors are more familiar with project conventions and usually have high social capital within the community (Bosu and Carver 2014; Bosu et al. 2017). In this context, a *core* developer is a member of the most active group of developers that has been contributing to the project for a longer time, whereas a *peripheral* developer is an occasional contributor (Crowston et al. 2006; Joblin et al. 2017a,b; Mockus et al. 2002; Terceiro et al. 2010). This *core–peripheral structure* can be inferred from analyzing developer interactions leveraging network analysis metrics in developer networks (Joblin et al. 2017a; Bock et al. 2023). This dichotomization (core vs. peripheral) of the importance of developers (Joblin et al. 2017a; Bosu and Carver 2014) is not ideally suited, though, to describe the continuously growing experience and standing of developers and can even lead to information loss (Fedorov et al. 2009). Taking this methodological issue into account, we employ the notion of *developer coreness* in our study, which is a continuous measure capturing the position of developers in the organizational hierarchy (Borgatti and Everett 2000; Christian and Vu 2021; Hechtl et al. 2025). Using this continuous measure, we investigate the received *feedback* on contributions and the *acceptance* of contributions through the code contribution process. We call this process the *integration process*, as patches (i.e., code contributions) need to be integrated into the code base. The feedback mechanism is a major means for quality control for the community, and the acceptance of a code submission is the desired result of every developer who contributes. These two factors represent the variables that can most likely be influenced by a developer's coreness, as suggested but only partially investigated in the literature (Bosu and Carver 2014).

To investigate the influence of coreness on these two variables (feedback/acceptance), we perform a mixed-method study on 16 popular OSS projects, which cover a wide range of different application domains, project sizes, and ages. This includes small and comparably young projects, such as JAILHOUSE, for which we analyze 8 years and about 2700 commits, as well as large and well-established projects, such as GCC, which is well over 30 years old and has over 134 000 commits. With our project selection, we aim at obtaining reasonably generalizable results and findings for projects that use a mailing list as their main contribution method. While mailing lists may be perceived as outdated in comparison to the more modern Web-based platforms (Zhu et al. 2016), numerous substantial and popular projects rely on mailing lists for patch submission, as recent research has shown (Mannan et al. 2020; Ramsauer et al. 2019; Käfer et al. 2018). These projects are usually long-lasting with well-established processes, and form a considerable foundation of today's global IT infrastructure, including the LINUX kernel, OPENSSL, and GCC.[1] Mining the histories of our 16 subject projects, we build *developer networks* to determine the coreness of developers and relate it to the feedback on and acceptance of contributions to determine the influence of the coreness on these aspects of the integration process.

The results of our *quantitative analysis* suggest that, while there are considerable differences across projects, *the coreness of a developer significantly influences the code contribution practice and success*. In particular, we find that developers with higher coreness values experience an, at large, quicker and more often successful review process for their code contributions. Regarding feedback developers receive

[1] https://kernel-recipes.org/en/2016/talks/patches-carved-into-stone-tablets/

on their contributions, we obtain a mixed picture: In some projects, developers with higher coreness values receive quicker feedback on their contributions, whereas in other projects, developers with lower coreness values are favored in regards to feedback. While our results corroborate previous results (Bosu and Carver 2014; Cetin 2019; Agrawal et al. 2018; Oliva et al. 2012) in that more experienced developers usually are assumed to have higher social standing in an OSS project community, our results provide novel and important insights into one of the most important processes of OSS development by shedding light on the relationship of developer coreness and the integration process. With our methodology, we provide a nuanced view on the standing of developers (i.e., coreness) and are thus able to provide a more detailed picture of the influence of the organizational hierarchy on the integration process.

To complement our quantitative analysis, we perform a *qualitative analysis* determining whether we can detect potential discrepancies in the integration processes of all our subject projects. In particular, we find that the absence of a link between developer coreness and the probabilities of receiving feedback and patch acceptance may hint at procedural problems in the development process of an OSS project. For triangulation, we conduct expert interviews with maintainers or core contributors of our subject projects, which confirm that unexpected results of our quantitative analyses (i.e., developers with higher coreness not having a higher chance of patch acceptance or feedback on their patches) can indeed be a sign of procedural problems in the integration process.

In summary, we make the following contributions:

- We propose a novel methodological framework that enables fine-grained analysis of the patch-review process in mailing-list-based OSS projects, integrating patch detection and mapping, socio-technical network construction, and the continuous measurement of developer coreness into a cohesive analysis pipeline.
- We collect and analyze data from 16 widely-used OSS projects to quantitatively investigate the influence of developer coreness on the feedback and acceptance of code contributions. In a nutshell, we found that developers with higher coreness values are favored when it comes to acceptance; for review feedback, this only holds for some projects.
- We perform a qualitative analysis including expert interviews with maintainers or core contributors of the analyzed projects to examine whether there are procedural problems in their projects. We found that, in most cases, in the projects that do not exhibit a correlation between developer coreness and the probability to receive feedback on or the acceptance of patches this does indeed indicate potential problems in the development process.
- We propose solutions to several recurring technical challenges that arise when analyzing patch data of OSS projects that use mailing lists as their major contribution system.
- ***Data Availability:*** We provide a replication package including the tools, scripts, and data of the analysis on our supplementary website: https://se-sic.github.io/paper-patch-survival-analysis/

2 State of the Art

In this section, we provide necessary background on the patch-review process and developer roles. Furthermore, we describe the current state of the art for research on the contribution processes of OSS projects and present related studies.

2.1 Patch-Review Process and Governance Structures

Code review is a key element of the software development process, and the artifacts arising from it can be used to infer roles of developers within a project. Many widely-used and established OSS projects use mailing lists for discussion, patch submission, and patch review on a regular basis (Käfer et al. 2018; Storey et al. 2017; Yin et al. 2023). These mailing lists are the central *communication hubs* in several globally distributed open-source projects (Guzzi et al. 2013; Corbet 2011). Developers create or edit source code on their local machine, and later send code changes (i.e., patches) to public lists, where anyone can join the discussion. Mailing lists are tried and trusted (Käfer et al. 2018). Although they seem outdated or low-level at first glance (Mannan et al. 2020; Jiang et al. 2014; Kroah-Hartman 2016), they are not only used in mature projects, but also in recent development undertakings, such as the secured communication tool WIREGUARD[2]. Another example of popular OSS projects that use mailing lists are projects in the APACHE environment (Storey et al. 2017; Yin et al. 2023). Archives for developer mailing lists are often publicly available and either directly provided by the project, or by secondary providers (such as GMANE[3]).

After a patch review, which is a discussion of involved developers on the list, maintainers (i.e., trusted individuals that are authorized to commit changes to the official source-code repository (Mauerer and Jaeger 2013; Rigby et al. 2008; Tan et al. 2020)) decide whether and how to integrate a code change in form of a patch. A patch can either generally be rejected for various reasons (e.g., the quality of the code is too low or the change is too complicated to maintain in the future), or the maintainer requests further modifications to the patch that are required before it can finally be integrated (Ramsauer et al. 2020; Rigby et al. 2008). Note that it is not untypical that this process of stepwise code refinement repeats several times (Wirth 1971; Ramsauer et al. 2019).

Furthermore, the patch-review process is heavily dependent on the governance structure of an OSS project (Izquierdo and Cabot 2015; Chulani et al. 2008). The governance structure of an OSS project usually consists of a set of rules or policies that describe how patches should be contributed to the project and how patches are accepted or rejected (Izquierdo and Cabot 2015; Markus 2007). While some projects explicitly define these rules and document them properly[4], others assume such rules without stating them explicitly (Izquierdo and Cabot 2015; Shibuya and Tamai 2009). Rigby et al. (2014) define the governance structure of OSS projects based on when and how patches are reviewed in the project. They differentiate between two main categories: *(1) Review-then-Commit (RTC)*, meaning a change is first sent for review

[2] https://www.wireguard.com/
[3] https://gmane.io/
[4] For example, the patch-submission policy of QEMU: https://www.qemu.org/docs/master/devel/submitting-a-patch.html

via, for example, the mailing list and is only integrated after the review; and *(2) Commit-then-Review (CTR)*, in which certain trusted users are allowed to integrate a change directly, which is then reviewed after integration. RTC is the most common approach in OSS projects allowing only already reviewed and approved changes to be integrated into the main repository. Nevertheless, CTR is also used and can be useful if only one or very few developers actually review changes as they can then directly test the changes in the main repository. While the term governance structure is used in different contexts in the literature and can also be used to describe the organizational structure of a project (De Laat 2007; O'Mahony 2007; Shaikh and Henfridsson 2017), we use it specifically in the context of the patch-review process to describe the rules and policies that govern this process in OSS projects.

Nurolahzade et al. (2009) investigate the patch-review process in the Mozilla Firefox project. They find that different reviewers handle patches differently, and while they seem to follow the rules of the review guidelines, the review is still dependent on the role of the reviewer. For example, module owners focus their reviews on code quality and long-term maintainability while peers (i.e., other developers in their modules) are more concerned with usability and functionality.

Rigby and Storey (2011) analyze the patch-review process in projects using a broadcast-based model for patch submission, such as in the Linux kernel. They find that, while it might be intuitive that a broadcast-based submission system might not be as efficient as a more targeted approach (e.g., writing responsible module owner directly), the broadcast-based system actually holds up well in terms of efficiency. One factor they identify as a reason for this is an adherence to the community norms and practices, which are effectively the governance structure of the project.

To compare different governance structures, Wang et al. (2015) analyze the patch-review processes of the Mozilla and Python communities. They find that the patch-review processes of these two communities differ significantly and conclude that there is not one universal solution for organizing peer review. So, the governing processes of OSS communities, while oftentimes similar, differ in their details and thus have to be analyzed with these differentiations in mind by researchers as well as the project communities.

2.2 Developer Roles

When studying the patch-review process, it is important to consider that OSS developers take on different roles that are usually not explicitly defined (Crowston et al. 2006; Acuna and Juristo 2004; Cheng et al. 2017; Tamburri et al. 2013). These roles can be inferred from the developers' activities in the project, though, are also a measure of a developer's experience and reputation in the project (Joblin et al. 2017a; Bosu and Carver 2014). This circumstance has been considered by multiple studies that have related these roles to the outcomes of the patch-review process and found that developers with higher reputation, for example, get their patches accepted faster (Bosu and Carver 2014; Cetin 2019; Agrawal et al. 2018; Oliva et al. 2012).

The importance and reputation of developers is not an entirely technical question, but depends also on social factors and other influences that are hard to quantify in practice. One common theme in the literature—in contrast to the approach taken

in this paper—is to dichotomize developer importance to end up with a relatively simple distinction of core and peripheral developer roles (Pinto et al. 2016; Cheng et al. 2017; Acuna and Juristo 2004; Yamashita et al. 2015; Torres et al. 2011; Joblin et al. 2017a). The classification of these roles is usually done by using network metrics on developer networks and splitting the resulting values at the 80th percentile with every developer above this threshold being classified as core and the rest as peripheral (Joblin et al. 2017a; Bock et al. 2023). The problem is that this dichotomization can lead to information loss (Fedorov et al. 2009) as the experience level of a developer (which we proxy with the notion of developer coreness) is not a binary value. Taking this into account, we operationalize developer coreness on a continuous scale in our study instead of dichotomizing developers into core and peripheral roles (Borgatti and Everett 2000; Christian and Vu 2021; Hechtl et al. 2025). With this continuous characterization of developers, we follow in the footsteps of previous work that has used more general indicators of developer experience or reputation, such as the centrality of a developer in the e-mail network, to find relationships between, for example, commit activity and coreness (or related measures) (Bird et al. 2006; Maruping et al. 2019; He et al. 2012; Shin et al. 2011).

2.3 Process Analysis and the Influence of Developer Roles

The patch-review process is a key step in the development of OSS projects and has been analyzed in various studies. As this process is inherently socio-technical, that is, the developers of the projects are involved in every step of the process, it is important to understand the influence of developers and their implicit roles on the patch-review process.

The processes that govern the development of OSS projects are usually mandated by coordination efforts of the community across the often globally distributed teams. Joblin et al. (2017b) state that the coordination and organization efforts are usually either mandated by some governing body (e.g., the Linux Foundation) or are self-organized by the community. Analyzing these organizational processes requires data that are usually mined from the projects' repositories, such as revision control logs or the release history (Gall et al. 1998; Zimmermann et al. 2005; Bird et al. 2006; Bock et al. 2023; Joblin et al. 2017a). Using these mined collaboration (commit) and communication (e-mails or issues) data is standard practice when analyzing various aspects of OSS projects, such as the patch-review process, the feedback and integration dynamics, or the developer reputation and roles within the project (Bock et al. 2023; Joblin et al. 2017a; Bosu and Carver 2014).

Two critical aspects of this *patch-review process* are the feedback dynamics as well as the integration of patches into a project's code base. The feedback dynamic is usually analyzed by looking at the quality of feedback that submitters of patches receive (Kononenko et al. 2016, 2015) or how feedback influences the software quality (McIntosh et al. 2016). Kononenko et al. (2015) investigate how the participation of developers in the feedback cycle of Mozilla influences the quality of the feedback. They find that ample participation as well as the experience of the reviewer influence the quality of the feedback positively. Khatoonabadi et al. (2023) find that the reputation of developers has a positive influence on the speed of the first feedback received

on a patch, which is in line with the findings of Bosu and Carver (2014). McIntosh et al. (2014, 2016) show that the feedback quality influences the software quality. So, it is important to understand *how* the feedback dynamics influence the patch-review process.

The integration of patches into the code base is another important aspect of the patch-review process, as this is the ultimate goal of developers submitting their patches for review. The integration mechanics and especially the integration speed and what affects it has been studied by Jiang et al. (2013). They find that multiple factors influence the integration speed, such as the size of the patch, the experience of the submitter, or the complexity of the patch. In the same vein, Bosu and Carver (2014) find that the experience of a developer influences the integration speed and success of their patches positively. Tan and Zhou (2019) find that effective communication by the patch submitter can also influence the integration success of a patch.

Bosu et al. (2017) report that code review, in general, is positively influenced (outcome and quality) by different factors: A good prior relationship between the reviewer and the submitter, a good reputation of the submitter, the expertise of the submitter in the concrete technical area of the patch, and whether the review of the code is considered 'low-effort'.

Although, in our study, we focus on the patch-review process that uses mailing lists for patch submission, there are similar processes in more modern social coding platforms. For example, GitHub uses pull requests for patch submission and supports code reviews via a Web interface providing developers with feedback and accept or reject their proposed changes. Researchers have conducted multiple studies to investigate feedback times and merge rates of pull requests (Alami et al. 2022; Zhang et al. 2023; Dey and Mockus 2020). These studies show that the size of patches, the communication of the submitter with the reviewers, and the use of continuous integration tools are among the key factors that can influence the patch-review process. Zhu et al. (2016) compare the contribution process in pull request-based systems with more traditional systems such as the mailing list and issue trackers. They find that, while pull request-based systems are overall more efficient than traditional systems, both have their advantages and disadvantages. One major difference when analyzing the patch-review process in projects that utilize mailing lists as compared to pull requests, is the mapping of contributions to their corresponding reviews. In pull-request based systems, the mapping of contributions to their corresponding reviews is straightforward, as the pull request is directly linked to the commit in the main repository. In mailing-list based systems, however, this mapping is not as straightforward. The reason is that the patches are usually sent to the mailing list as a plain text e-mail, and the commit in the main repository is created after the patch has been accepted by the maintainer. This makes the analysis of the patch-review process more difficult (Jiang et al. 2014; Ramsauer et al. 2019), which we describe in detail in Section 4.2. Nevertheless, as mailing lists are still widely used, they are an important part of the OSS ecosystem and hold valuable insights. We focus on the peculiarities of the mailing-list based patch-review process in our study.

2.4 Survival Analysis

When analyzing the collaboration process of OSS projects, there are many factors that have to be considered. One of these factors is the statistical method used to analyze the collaboration and communication data. Qiu et al. (2019), for example, use survival analysis to investigate whether the social capital of developers influences their continued participation in OSS projects. Also, Ait et al. (2022) use survival analysis to investigate whether OSS projects fail or continue to exist after a certain time.

Survival analysis was originally introduced in the field of medicine. It is usually used to determine the probability of a patient surviving under the influence of some other factors, such as a certain disease or medication (Rich et al. 2010; Kleinbaum and Klein 2012). This is done by measuring the time it takes until a certain event takes place (usually the death of a patient). Researchers can analyze this time and correlate it to influencing factors, such as in a two-group medical study where one group is given a new drug and the other group is given a placebo. With such a study, researchers can identify the influence that either of the treatment options have on the risk of the patients' death.

Survival analysis has found wide applicability in other research fields today, including software engineering. Lin et al. (2017) utilizes survival analysis to investigate the factors that lead to developer turnover in OSS projects. They find that developers that join a project in the early stages of its lifetime mainly modify existing files, and mainly write code as opposed to documentation. These developers have an overall lower turnover rate.

Ortega and Izquierdo-Cortazar (2009) use survival analysis to approximate the time a developer stays with an OSS project. They find that, on average, the retention period for developers in such projects lies between 500 and 1000 days. Aman et al. (2017) investigate the survival times of bugs in code files. They find that bugs that were introduced by new developers have about 26% shorter survival than bugs introduced by long-term developers.

One specific type of survival analysis is *Cox proportional hazard analysis* (Cox and Oakes 1984). This is a form of regression analysis to calculate the instantaneous risk of a subject to experience the event at any given point in time. This risk is called the *hazard ratio*. A higher hazard ratio means that a subject has a higher probability of experiencing the event at any point in time.

Cox proportional hazard analysis has been used in OSS research before. Bird et al. (2007) use the hazard ratio to investigate the factors that lead to people entering OSS projects. They find that technical skill and social reputation have a positive effect on the probability of a person becoming a developer in an OSS project. Using hazard ratios Koru et al. (2007) find that the size of a source code module in the Mozilla project has an effect on the defect proneness of the module.

In our study, we use the hazard ratio to obtain a quantitative measure of the influence of the coreness of developers on the patch-review process. We describe this in detail in Section 4.5.

3 Problem Statement and Research Questions

While the influence of developer roles on the patch-review process has been studied in the past, as presented in Section 2, it is still an important topic to investigate: The patch-review process is a key element in the development cycles of OSS projects that ensures code quality and is highly dependent on the roles developers take in the project; their adherence to the process as well as other factors might thus affect the overall sustainability of the project. Lots of studies in the field use different methodologies to analyze the role of developers in OSS projects (e.g., using different classification methods to determine the role of developers (Pinto et al. 2016; Cheng et al. 2017; Acuna and Juristo 2004; Yamashita et al. 2015; Torres et al. 2011; Joblin et al. 2017a)). It is not unreasonable to assume that the results of previous studies might be dependent on methodological details (e.g., with respect to developer classification) and consideration of various other factors of the analyses. To triangulate previous findings, we devise a new methodological approach that deals with recurring technical challenges in order to analyze the patch-review process in a more fine-grained manner than previous studies, allowing us to investigate whether previous findings still hold when using a more sophisticated research methodology. For this purpose, we set out to answer the following research questions, based on previous studies in the field (Bosu and Carver 2014; Bosu et al. 2017; Jiang et al. 2013) and adapted to our research methodology (which we describe in detail in Section 4):

RQ₁ *Does a higher coreness of developers increase the probability of the community to give feedback on their code submissions?*

RQ₂ *Does a higher coreness of developers increase the probability of successfully integrating their code submissions?*

On the basis of the quantitative analyses of RQ_1 and RQ_2, we conduct an in-depth qualitative analysis of the underlying patch-review process. In particular, we suspect that peculiar results that deviate from previous findings could indicate procedural problems in the patch-review process of individual subject projects. This is not unfounded, since Liu et al. (2024) have found that process anomalies in pull-request reviews of OSS projects can be indicators of suspicious incidents in the project. In this vein, we hypothesize that unexpected results in the analysis of the patch-review process can be indicators of procedural problems in OSS projects, such as missing reviews of critical parts of the source code. Therefore, we perform a pattern and outlier analysis based on the results of our first two research questions to gain a deeper understanding of the patch-review process in the subject projects. This leads us to the following research question:

RQ₃ *Do unexpected results in our quantitative analysis of the integration process of OSS projects in RQ_1 and RQ_2 indicate procedural problems of the development process?*

4 Methodology

In this section, we explain our research methodology. First, we describe our subject projects and how we extract and process commit and e-mail data. Second, we introduce how we construct developer networks from the data and how to determine developers' coreness. Thereafter, we explain how exactly we apply the Cox proportional hazard model. Last, we explain our methodology for the identification of process-deviant projects, our qualitative analysis for RQ_3, and our expert interviews.

As mentioned in Section 3, we utilize a more fine-grained methodology than previous work to analyze the patch-review process in OSS projects and take technical subtleties into account. In particular, we want to highlight three key differences to previous work: (1) We use the continuous measure of developer coreness instead of a strictly binary classification of a developer's position in the organizational hierarchy of OSS projects. This way, we factor in that the position is not a binary value but a continuous property of a developer. (2) We use survival analysis to gain a more detailed statistical insight into how the coreness of a developer influences the patch-review process at any point in time (which has not been done to study the patch-review process). (3) We analyze OSS projects that use a mailing list as their main communication and code review platform. We deliberately choose these projects to analyze the patch-review process in a more mature and long-lasting review environment. In the following, we take a closer look at the various details of our methodology.

4.1 Subject Projects

In this study, we aim at understanding long-lasting foundational OSS projects that have been active for many years. To do so, we choose 16 technically mature projects that provide traceable data on their development history over decades. Instead of transitory recent phenomena of the modern age, such as Web-based collaboration platforms (e.g., GitHub), we analyze outlasting projects that consequently bank on mailing lists as their main mode of communication as they are still a relevant and highly used means of developer communication as we have pointed out in Section 2.1. The projects we selected that fit into these criteria are Buildroot, BusyBox, Coreutils, DPDK, FFmpeg, FLAC, GCC, Git, Jailhouse, LLVM, OpenEmbedded-Core, OpenVPN, Postgres, QEMU, U-Boot, and Wine. Following the approach of Rigby et al. (2014), we first inspected the type of review processes that our subject projects use, to obtain an understanding of the processes used in these projects and to ensure that our methodology and especially our qualitative analysis of the projects are applicable. We did so by manually reading and analyzing the contribution guidelines of the projects[5]. All of them state that developers are required to send their patches to the project's mailing list for review before they can be integrated into the main repository. This shows that all of our subject projects use the *Review-then-Commit (RTC)* process, which is one of the most common review process forms in OSS projects (Rigby et al. 2014). Knowing this helps us to understand the results of our analyses and to pinpoint potential procedural problems in the integration processes of the projects.

[5]For example, the patch-submission policy of QEMU: https://www.qemu.org/docs/master/devel/submitting-a-patch.html

4.2 Data Extraction and Processing

To acquire and analyze the data of these 16 OSS projects, we use CODEFACE[6] (Joblin et al. 2015, 2017a,b) and its companion tool CODEFACE-EXTRACTION[7] to extract the commit metadata from the version control system (VCS) of the project. These include author name, e-mail address, and committer date for each commit. For extracting mailing-list data, we downloaded the project's development mailing list(s) (see Table 1) from the mailing-list archive GMANE[8] and organize metadata of e-mails, such as author name, e-mail address, and date via CODEFACE.

Developers may use multiple identities (i.e., different combinations of name and e-mail addresses) when contributing to an OSS project. As a countermeasure, we use the disambiguation heuristic of (Oliva et al. 2012) to harmonize different combinations, which has been proved to be accurate (Wiese et al. 2016).

Furthermore, we assign patches on mailing lists to related commits in the VCS. While mailing-list archives allow for

Table 1: The development mailing lists that contain patches for our subject projects.

Subject Project	Developer Mailing Lists Containing Patches
BUILDROOT	`gmane.comp.lib.uclibc.buildroot`*
BUSYBOX	`gmane.linux.busybox`
COREUTILS	`gmane.comp.gnu.coreutils.general`, `gmane.comp.gnu.core-utils.bugs`
DPDK	`gmane.comp.networking.dpdk.devel`
FFMPEG	`gmane.comp.video.ffmpeg.devel`
FLAC	`gmane.comp.audio.compression.flac.devel`
GCC	`gmane.comp.gcc.devel`, `gmane.comp.gcc.patches`
GIT	`gmane.comp.version-control.git`
JAILHOUSE	`gmane.linux.jailhouse`
LLVM	`gmane.comp.compilers.llvm.devel`, `gmane.comp.compilers.llvm.cvs`*,**
OPENEMBEDDED-CORE	`gmane.comp.handhelds.openembedded.core`
OPENVPN	`gmane.network.openvpn.devel`
POSTGRES	`gmane.comp.db.postgresql.devel.general`, `gmane.comp.db.postgresql.devel.patches`
QEMU	`gmane.comp.emulators.qemu`
U-BOOT	`gmane.comp.boot-loaders.u-boot`
WINE	`gmane.comp.emulators.wine.devel`, `gmane.comp.emulators.wine.patches`

* Contains not only patches submitted for revision, but also automated commit e-mails containing the applied patch; we have removed the automated commit e-mails for our analyses based on filtering subject prefixes (see Appendix A).

** Also known as `llvm-commits`, contains also automated commit e-mails and not only patches (see *).

tracking communication between participants within a discussion (i.e., by traversing e-mail threads), there are no built-in means to track the connection between (a) different

[6] See https://github.com/lfd/codeface/

[7] See https://github.com/se-sic/codeface-extraction/

[8] See https://gmane.io/. The front-end of GMANE is down at the time of writing, but the backend is accessible via NNTP protocol, for which we used the tool NNTP2MBOX (https://github.com/xai/nntp2mbox/) to extract the mbox files containing the e-mails form the mailing list(s).

revisions of a patch within the mailing list, and (b) the connection between the latest revision of the patch and the resulting commit in the repository (Jiang et al. 2013; Ramsauer et al. 2019). Especially, when patches require several major revisions over time before they are finally integrated, they may significantly differ from the version in the repository. Therefore, it is a non-trivial challenge to reconstruct the connection of patches on mailing lists and the repository (Ramsauer et al. 2019; Jiang et al. 2014).

To reconstruct these connections, we use the tool `PaStA`[9], which identifies patches on mailing lists and systematically maps them against commits in the repository. As described by Ramsauer et al. (2019), `PaStA` uses heuristics to detect semantically similar patches. `PaStA` reconstructs *patch-revision sets*, that is, semantically similar patches and commits. As our data set dates back years or even decades (see Table 2), we assume that every accepted patch submitted to the mailing list will show up in the VCS (except for most recent patches which are not finally integrated). Hence, if a patch (i.e., one e-mail in the patch-revision set that contains the patch) is mapped to a commit, we consider the patch *accepted*. Otherwise, if a patch-revision set is not mapped to any commit in the VCS, we consider the patch *rejected*.

We have performed multiple sanity checks and have undertaken extensive cleanup steps on our data, enabling `PaStA` to also identify patches that use uncommon patch formats, etc., on the mailing lists. In a similar vein, we removed e-mails that look like patches but actually are automatic commit e-mails after the commits have been integrated into the repository. We describe the details about these technical challenges in Appendix A. Regarding `PaStA`, Ramsauer et al. (2019) demonstrated the high accuracy of the tool: Using a sample of 1047 e-mails from a Linux kernel mailing list, they reach a Fowlkes-Mallows[10] index of 0.988, indicating that the tool attains both high precision and high recall.

When investigating whether a developer received feedback on their submitted patch, we search for e-mails that meet all of the following criteria:

1. They belong to the same e-mail thread as the submitted patch.
2. They were sent after the patch was submitted.
3. They were sent by a developer other than the patch submitter.

If there is no such e-mail on the mailing list fulfilling these requirements, we assume that there is no feedback for this patch. While this approach does not give us a guarantee that the identified e-mail is an actual reply to the submitted patch, the fact that it is part of the same e-mail thread increases confidence in this assumption. An additional sanity check by sampling 10 patch-reply pairs from each of our 16 subject projects (160 pairs in total) and manually checking whether these are actual replies to the previously sent patch exemplarily confirms our assumption: Of the 160 samples, only 7 replies were not related to the previously sent patch, which is less than 5%. Consequently, we are confident that our heuristic is viable.

[9] See https://github.com/lfd/PaStA/

[10] The Fowlkes-Mallows index measures the similarity between two clusterings of the same data set by computing the geometric mean of pairwise precision and recall, ranging from 0 (no agreement) to 1 (perfect agreement) (Fowlkes and Mallows 1983).

4.3 Developer Network Construction and Coreness Calculation

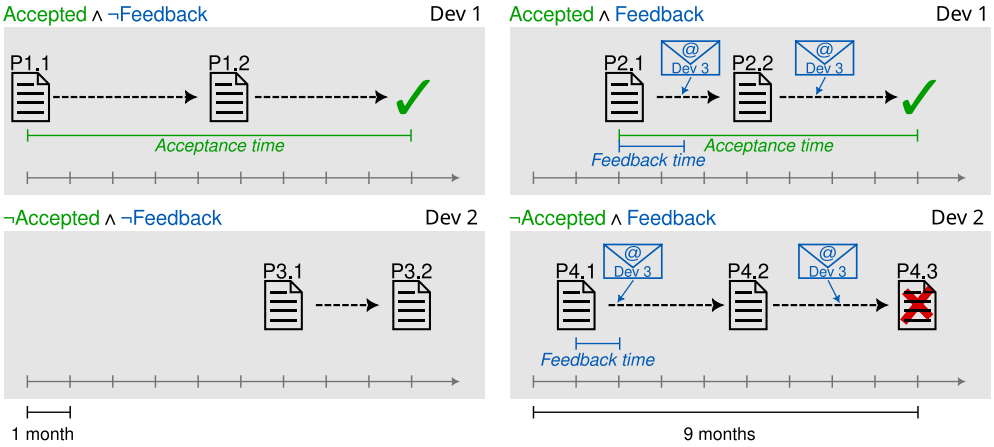

Fig. 1: Overview of the four different scenarios for a submitted patch in the integration process. The patch can either be accepted or not, and it can receive feedback or not. The 9-months time window prior to patch acceptance or prior to the last patch revision in case of not-acceptance (indicated by the timeline at the bottom of each scenario) is used for determining the developer's coreness for the respective patch submission. The feedback time and acceptance time are the times that we use for the statistical analysis and represent the time it takes for a patch to be accepted (see developer Dev 1) or a developer (i.e., Dev 1 or Dev 2) to receive feedback on their submission by another developer (i.e., Dev 3).

To answer our research questions, we determine the coreness value of every developer at the time of each of their patches' submission. In Figure 1, we provide an overview of four different scenarios in terms of four exemplary patch-revision sets.

First of all, for each patch-revision set, we only consider the author of the first e-mail related to the set, as we base our analysis on the initial submission of a patch. In the patch-revision sets in Figure 1, this would be the patches P1.1, P2.1, P3.1, and P4.1. By doing that, we lose potential submissions of revised patches by others than the initial developer of that patch. However, we investigated how often there are patch-revision sets in which different developers sent revisions of the same patch. Our investigations show that we can neglect this case as it usually occurs in less than 5% of the detected patch-revision sets (except for project QEMU where it occurs in 33%; we could explain this phenomenon by an active reviewer community that often takes a submitted patch, makes changes to it, and posts the improved version back into the thread).

As a developer's coreness value may change throughout project evolution, we need to determine the coreness separately for each patch (i.e., patch-revision set). Therefore, we create an individual 9-months developer network for each patch, ending at the time of patch integration into the repository (i.e., the committer date of an accepted patch), or—if the patch was not accepted—the submission date of the last patch revision. We show this for the four different scenarios via the four timelines in Figure 1. Hence, we consider the last 9 months before a patch might get accepted/submitted to determine its developer's coreness. We also conducted sensitivity analyses using

windows of 3 months, 6 months, and 12 months. A window of 9 months emerged as a reasonable choice, as shorter windows introduce spurious patterns due to high fluctuations in coreness values, while longer windows risk capturing outdated values given that a developer's coreness can change considerably over the course of a year. Our decision for 9-months windows is also motivated by the fact that some patch reviews lead to many discussions and revisions, taking several months until a patch gets accepted. According to the literature (Jiang et al. 2013), most of the accepted patches should be accepted within 9 months after initial submission.

We associate the coreness of a developer with the centrality of the developer in the community (Christian and Vu 2021). Based on the 9-months window determined for each patch individually, we build three different kinds of undirected, unsimplified developer networks (i.e., the developers are represented by the vertices in the network and the edges represent interactions between developers) to handle different perspectives of centrality in the community:

cochange To capture the centrality within the technical part of the project, we build cochange networks, where two developers are connected by an edge when they have edited the same file within the respective time window (Zimmermann et al. 2005; Gall et al. 1998; Joblin et al. 2015).

mail To capture the centrality within the communication structure of the project, we build mail networks, in which two developers are connected by an edge when they have replied to the same mailing-list thread within the respective time window (Bird et al. 2006; Storey et al. 2017).

both To obtain a holistic view on socio-technical aspects of the project, we combine *cochange* and *mail* networks into a single network covering both technical collaboration and communication activity between developers (Joblin et al. 2023; Hechtl et al. 2025).

Finally, we use two different network centrality metrics to determine the coreness of a developer: eigenvector centrality and hierarchy centrality. This way we get two different perspectives on the coreness of developers, as the two metrics model different properties to determine the centrality. Using multiple different centrality metrics to model different perspectives is a known concept in the literature (Bock et al. 2023; Hechtl et al. 2025). In the following, we describe the two metrics in more detail and explain why we use these two metrics in particular.

Eigenvector centrality infers the centrality of a vertex in a network by determining the centralities of the vertices in its direct neighborhood. If a vertex is embedded in a high-centrality neighborhood, the vertex itself gets a higher centrality value. The formula to compute the eigenvector centrality x_i for developer i is:

$$x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j \tag{1}$$

$N(i)$ describes the set of all neighbors of vertex i and λ is a proportionality constant (Joblin et al. 2017a; Brandes and Erlebach 2005). Under this metric, developers

receive high centrality values if they are connected to other individuals with high centrality. In other words, if a developer mostly communicates (mail network) or collaborates (cochange network) with developers that have high centrality values themselves, the developer gets a high centrality value as well.

Hierarchy centrality is a metric that describes how local neighborhoods are organized among themselves. This is achieved by combining two concepts: *vertex degree* and *clustering coefficient*. The vertex degree describes how many edges are connected to a vertex. The clustering coefficient is used to measure how embedded a vertex is within a cluster of vertices. The formula of the clustering coefficient of a vertex i is

$$c_i = \frac{2n_i}{k_i(k_i - 1)} \tag{2}$$

where k_i is the number of edges connected to vertex i, and n_i is the number of edges between the neighbors of i (Boccaletti et al. 2006). The hierarchy centrality of a vertex is then determined using these two metrics: The higher the vertex degree and the lower the clustering coefficient, the higher the hierarchy centrality value (Joblin et al. 2017a, 2023). The underlying theory for receiving a higher centrality value with a lower clustering coefficient and a higher vertex degree is that these developers communicate (mail network) or collaborate (cochange network) with many different developers from different local clusters while not being embedded in a local cluster of developers, thus being more of a "hub" in the network.

While there are multiple possible network metrics that can be used to determine the centrality of a developer in a network, we have chosen these two metrics as they are well-established in the literature and have been shown to be useful for determining the coreness of developers in OSS projects (Joblin et al. 2017a, 2023; Bock et al. 2023). We use these metrics specifically, because they model the centrality of developers on different scales. Eigenvector centrality models a global view of the network by considering the centrality of the neighbors of a developer, while hierarchy centrality considers the community structure of the network.

To sum up, we have the network type and coreness metrics as two variables: We use three different network types (*cochange*, *mail*, *both*) together with two different coreness metrics (*eigenvector centrality*, *hierarchy centrality*). In our empirical study, we refer to the combination of network type and coreness metric as *configuration*. To preprocess our data, build the networks, and calculate the coreness metrics, we use the network library CORONET[11] and additional scripts. All in all, we end up having six configurations per project, which enables us to have different points of view on our subject projects. The three network types cover the technical side, the social side, and a holistic view of both sides combined for each project. The two coreness metrics provide different views of a developer's positioning in the organizational structure of the project—either being connected to many very central developers (eigenvector centrality), or being a hub in the hierarchical structure (hierarchy centrality). By considering all six configurations, we aim to obtain a more comprehensive understanding of the relationship between developer coreness and the patch-review process.

[11] See https://github.com/se-sic/coronet/

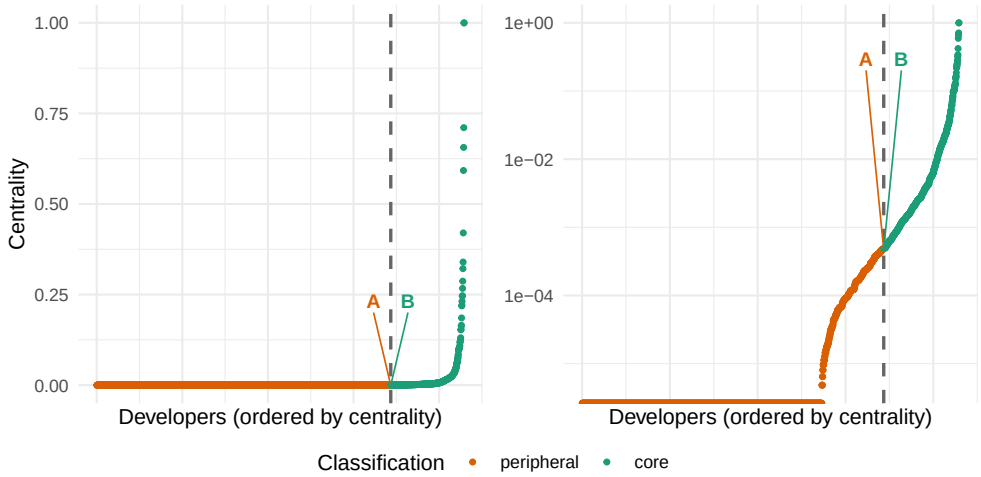

Fig. 2: Scatter plots of the distribution of centrality values of developers in the cochange network of DPDK using hierarchy centrality. The left plot shows the values on a decimal y-scale from 0 to 1, while the plot on the right shows the same values on a logarithmic scale. The dashed line shows the threshold for a binary core/periphery classification and the colors indicate the role a developer would be classified into. The points labeled with "A" and "B" show two developers with very similar centrality values, but that would be classified into different roles.

4.4 Coreness vs. Core/Periphery Classification

As described in Section 4.3, we employ the continuous measure of developer coreness instead of the classic core/periphery classification to determine the position of developers in the organizational hierarchy of OSS projects. The binary classification of developers into core and peripheral developers is a common approach in the literature (Crowston et al. 2006; Joblin et al. 2017a; Bock et al. 2023), but it has some limitations. To illustrate these limitations, let us use an example: In Figure 2, we show the distribution of the hierarchy-centrality values of developers in DPDK measured on the cochange network. On the left side, we show the values on a decimal scale, and, on the right, on a logarithmic scale. The dashed line indicates the 80% threshold that would be used to classify developers into core and peripheral developers. When comparing the developers whose centrality values are close to the threshold, we can see that there are developers with very similar values on both sides of the threshold. Taking for example the developers labeled with "A" and "B" in Figure 2, we can see that they have very similar centrality values, but developer A would be classified as a core developer while developer B would be classified as a peripheral developer. Furthermore, when looking at the left plot, we can see that a chunk of the developers that would get classified as core developers have very low centrality values, while developers with only minimally lower values are classified as peripheral developers. These imperfections of the core/periphery structure are the reason why we use the continuous measure of developer coreness instead of a binary classification of developers into core and peripheral developers.

4.5 Cox Proportional Hazard Model

Having determined the coreness of developers in our subject projects, we employ the Cox proportional hazard model (see Section 2.4) to answer our first two research questions. We do so to gain insights into the influence of the coreness of developers on the probability of receiving feedback on their patch submissions and the probability of getting their patches accepted. This model gives us unique and novel insights into the relation of developer coreness and its relation to the patch-review process as it takes time and the order of events into account.

The hazard ratio of the model is calculated as follows:

$$h(t, X) = h_0(t) \cdot \mathrm{e}^{\sum_{i=1}^{p} \beta_i X_i} \tag{3}$$

where $h(t, X)$ describes the hazard at time t under the influence of the variables X. The term $h_0(t)$ describes the base hazard at time t without the influence of any variable, and the exponential term describes the sum of all influencing variables multiplied by a regression coefficient β (Kleinbaum and Klein 2012). If the event is, for example, the acceptance of a patch, then a higher hazard ratio means that a patch has a higher chance of being accepted at any given point in time than a patch with a lower hazard ratio. As described in Section 4.3 we use six different configurations (three network types and two coreness metrics) to determine the coreness of developers and analyze each configuration separately with the Cox proportional hazard model for the first two research questions.

To analyze RQ_1 (i.e., whether the coreness of developers influences the probability of receiving feedback on their code submissions), we have to define the event that is used for the Cox proportional hazard analysis. In this case, the event is a developer getting feedback on a patch submission. For each patch-revision set, we determine whether this event happens, and if so, how long it took. Then, we can analyze the resulting data with the Cox proportional hazard model to calculate the overall hazard ratio of the project and configuration in question. A hazard ratio larger than 1 means that developers with higher coreness values have a higher probability to get feedback on patch submissions than developers with lower coreness values at any given time from submission of the patch until the event happens. A hazard ratio lower than 1 shows the exact opposite. We expect that developers with higher coreness values should receive feedback faster and more consistently than developers with lower coreness values. If this is not the case, we call this result *unexpected*. This is due to the conjecture that developers with higher coreness values are more integrated into the project communities and have therefore more social capital (Bosu and Carver 2014; Bosu et al. 2017). However, we admit that there are also plausible reasons that speak against our expectation. So, in contrast to our expectation, it could also be that developers with lower coreness values get more and faster feedback. This could be due to project communities wanting to value and bind new developers to their project, or simply because developers with lower coreness values are usually less experienced in the project and therefore need more feedback to fulfill the project's standards. Consequently, there are reasonable explanations for both situations, resulting in hazard ratios being above 1 if developers

with higher coreness preferably receive feedback or below 1 if developers with lower coreness preferably receive feedback.

To analyze RQ_2 (i.e., whether the coreness of developers influences the probability of getting their patches successfully integrated), we proceed in a similar way. The event that is used for the Cox proportional hazard analysis, in this case, is the commit with which the patch is integrated into the VCS of the subject project. Again, a hazard ratio above 1 indicates that developers with higher coreness values have a higher chance of patch acceptance at any given point in time than developers with lower coreness values. A hazard ratio lower than 1 shows the opposite. We expect that developers with higher coreness values have a higher probability to get their patches accepted than developers with lower coreness values. If this is not the case, we call this result *unexpected*. This is due to the conjecture developers with higher coreness values have more experience with the code base of the project and therefore know how to write patches that are necessary and high-quality.

Clearly, beside the developer coreness, also other factors could influence the probability of getting a patch accepted. In particular, the number of reviewers (Jiang et al. 2013) (i.e., the number of developers that have reviewed a patch) could influence the probability of getting a patch accepted, as a higher number of reviewers could lead to a higher agreement with the proposed patch, or, in contrast, could indicate a high number of issues identified in the proposed patch. Therefore, we add the number of reviewers of a patch as an additional factor (i.e., covariate) to the Cox proportional hazard model of RQ2. We measure the number of reviewers by counting the distinct developers that have commented on (i.e., replied to) a patch on the mailing list. We add this additional factor to the Cox proportional hazard model of RQ_2. This way, we obtain separate hazard ratios for the developer coreness and the number of reviewers. These hazard ratios (while being separate) are however not independent of each other and reflect the influence of the factors on the probability of getting a patch accepted at any given point in time under the influence of the other factor. This allows us to analyze whether the coreness of a developer has an influence on the probability of getting a patch accepted, even despite the number of reviewers.

4.6 Process-Deviant Projects and Qualitative Analysis

To obtain a deeper understanding of the concrete processes used in our subject projects, we perform a qualitative pattern and outlier analysis of the results of RQ_1 (feedback) and RQ_2 (patch acceptance). In particular, we investigate in RQ_3 whether unexpected results can indicate procedural problems in a project's patch-review process or whether we miss important points that are not covered by our analysis. Given the diversity of our subject projects and the different ways how the projects particularly implement the integration process, there is no strict universal mechanism to definitively determine potential procedural problems. Since feedback and patch acceptance, while being the central parts of the patch-review process, are not the only parts of the process, we also consider other factors beyond the analyses of RQ_1 and RQ_2 to identify potential procedural problems in the projects. We consider the following three criteria to identify whether a project is *process-deviant* (i.e., a project that violates the following points), or *process-conforming* (i.e., a project that fulfills the following points):

1. We take a look at the *commit coverage*. The commit coverage is the percentage of commits in the project's VCS that can be mapped to patches on the mailing list. If this coverage is low, then this is an indicator that either a lot of code contributions had not been previously sent to and reviewed on the mailing list (potentially violating the project's contribution policy), or that there are other technical difficulties that prevent the detection of patches on the mailing list. In a study of the Linux kernel, Ramsauer et al. (2020) show that they are able to map around 96% of the commits in the VCS to patches on the mailing list, which shows that the commit coverage can be very high if the patch-review process is well-defined and followed by the developers. In this case, the Linux kernel is an exceptional example, as it commonly takes the process adherence very seriously and have a very strict patch-review process. Furthermore, the kernel developers send almost all of their patches in a standardized format, which makes it easier to detect patches on the mailing list and to map them to commits in the VCS. In projects that do not enforce the patch-review process in all details as strictly as the Linux kernel but still have a well-defined process, we would still expect that a majority of the commits in the VCS can be mapped to patches on the mailing list, as the majority of the code contributions should be sent to the mailing list for review before they get integrated into the VCS. However, accounting for some possible irregularities in the process and in the mapping heuristic, we consider a commit coverage of around 2/3 (66.7%) to be a reasonable threshold for a project to be considered process-deviant. Any less than that leads to our analysis missing a substantial percentage of commits, which puts the strictness of the patch-review process in the project into question and diminishes the significance and informative value of our analyses with respect to RQ_1 and RQ_2. To further justify the threshold of 2/3, we also compare it against the average commit coverage of our subject projects and also check whether there is a considerable gap between the projects that are above and below this threshold. We consider projects with a commit coverage below our threshold to be process-deviant.
2. We look at the *statistical significance and direction of the results of the Cox proportional hazard analyses for RQ_2*. As described in the previous section, we expect the hazard ratio to be above 1, meaning that developers with higher coreness values have a higher probability of getting their patches accepted than developers with lower coreness values. If we find that a majority of our investigated configurations for a project are not statistically significant above 1, this is a strong indicator that a project could be process-deviant.
3. We look at the *statistical significance of the results of the Cox proportional hazard analyses for RQ_1*. As described in the previous section, we expect the majority of the configurations to be statistically significant above 1, meaning that developers with higher coreness values have a higher probability of getting feedback. However, in the previous section, we also have laid out reasonable explanations for why this expectation might not be fulfilled. Consequently, in contrast to RQ_2, if our expectation for RQ_1 does not hold for a project, this is not a strong indicator for a process-deviant project. Instead, we consider a project that, for the majority

of our configurations, shows statistically significant hazard ratios below 1 or a mixed picture not to be process-deviant.

In summary, we define a project as process-deviant if the commit coverage is low (less than the average) or if the results of RQ_2 do not follow our expectations. If only the results of RQ_1 do not reflect our expectations but the commit coverage is high ($\geq 66.7\%$) and the results of RQ_2 do reflect our expectations, then we do not consider these projects to be process-deviant.

To investigate whether there are procedural problems in the integration process of the projects that we consider to be process-deviant, we perform a qualitative analysis that helps us answer RQ_3. While we focus on these process-deviant projects in the qualitative analysis, we also analyze the process-conforming projects. This allows us to gain a deeper understanding if procedural problems only occur in the process-deviant projects or if they are also present in other projects. We perform the qualitative analysis in the following way:

First, we check whether there are only few developers that are consistently contributing to the project. We expect these developers to be at a higher risk not to follow the patch-review process (i.e., integrating the patch without review and without sending the patch to the mailing list prior to integration), as they are the ones that do most of the reviewing themselves. Consequently, a very low number of active developers might lead to unexpected results in our analyses, as patches that were never sent to the mailing list are not considered in our analyses of RQ_1 and RQ_2.

If we consider a project to be process-deviant because of its low commit coverage, we have to assume that many patches do not get reviewed (at least, not in the intended way) and thus do not undergo quality control. Hence, for projects with low commit coverage, we investigate commits in the projects' VCS that could not be mapped to patches on the mailing list. For our qualitative analysis, we sample some of these commits for each of the affected projects to further look into. Specifically, we check whether the commits were wrongfully not mapped (because of uncertainties of our mapping strategy) or whether these commits are trivial patches, such as changes to copyright headers or `README` files. Moreover, we manually analyze who the developers responsible for the un-mapped commits are, and whether we can find a pattern there. In addition, we check whether the commits that were not mapped to patches on the mailing list were integrated from other sources (e.g., other repositories or other communication channels), as this shows that the integration process is either not well-defined or not followed by some developers. This can lead to confusion about the set integration process and can also lead to un-reviewed code entering the project.

We deduce that there are indeed procedural problems in the integration process of a project if there are a lot of patches that get pushed directly to the project's VCS without review. This shows that some developers do not follow the defined integration process and can lead to bugs and security issues if un-reviewed patches are integrated into the source code of the project without proper quality control (Michlmayr et al. 2005; Haider et al. 2023).

4.7 Semi-Structured Interviews of Core Contributors

Subsequent to the qualitative analysis of the projects, we attempt to contact a few maintainers and core contributors of each of our 16 subject projects (i.e., not only the process-deviant projects, but also all other projects) to gain insights into their integration processes. For this purpose, we first manually identify these developers in the projects by examining project-reported data for maintainers and the top contributors in the commit data of the projects. Of these developers, we manually select the ones that are currently still contributing and have been with the project for a longer time (i.e., multiple years) to ensure that they have lots of experience in the project. While this manual selection process is subjective, we put in a lot of effort to ensure that we select the most suitable and experienced developers in the projects. We then reach out to these developers (between one and three per project) via e-mail to ask them questions about the integration process of their project. We formulate different e-mails for the projects that we have identified as process-deviant (based on the results of our analyses for RQ_1 and RQ_2) and the projects that we consider process-conform. For the projects that we consider to be process-conform, we ask the interviewees about their integration process and whether they actively enforce their integration process. For the projects that we have identified as process-deviant, we ask the interviewees about the integration process, in general, and whether they have observed any procedural problems in the integration process. The concrete wording of our e-mails can be found on our supplementary website[12]. Depending on the responses that we receive from the developers via e-mail, we also follow up with additional questions via e-mail to gain a deeper understanding of the integration process of the projects if something is unclear from the initial response. We analyze the responses of the developers manually and have discussed them extensively in our team.

By conducting these interviews, we aim at gaining insights into the integration process of the individual projects and into the reasons for the observed discrepancies in the projects that we identify as process-deviant.

5 Descriptive Statistics

Using the data extraction and processing procedure described in Section 4.2, we extracted and processed data for the 16 projects introduced in Section 4.1. In this section, we discuss our project selection and present descriptive statistics about these projects to contextualize the results of our analyses in Section 6. In large, our project selection was guided by four main criteria: (1) the projects are from different domains, (2) the projects differ in size and age, (3) the projects use mailing lists for their patch-review process to follow the RTC model (see Section 2.1), and (4) the projects have a large enough number of patches and patch-revision sets to be able to perform a quantitative analysis. However, as data extraction and processing is computationally expensive, analyzing very large projects that fulfill our four main criteria, such as the Linux kernel, would not be feasible as such an analysis could take multiple months to complete. With these criteria in mind, we selected the 16 projects listed in Table 2. The mentioned difference in size and age between the projects can be seen in Table 2.

[12] https://se-sic.github.io/paper-patch-survival-analysis/

The larger projects, such as LLVM, have hundreds of thousands commits, e-mails, and patches, while the smaller projects, such as FLAC, only have a couple of thousands or hundreds. All of our projects use either C or C++ as their main programming language, except for Buildroot, which consists of mainly shell scripts and Python code while using only little C or C++.

Table 2: Numbers of the extracted commits, e-mails, authors (mail senders and commit authors), patches, patch-revision sets, and analyzed time period for each subject project.

Project	# commits	# e-mails	# authors	# patches	# patch-revision sets	time period
Buildroot	51 957	281 700	3 218	77 131	56 892	2006–2020
BusyBox	16 533	46 163	3 104	4 425	3 762	2002–2020
Coreutils	2 889	40 921	4 368	6 859	2 889	2010–2020
DPDK	25 177	190 555	2 146	79 803	25 117	2013–2020
FFmpeg	86 588	281 381	6 470	66 142	47 034	2003–2020
FLAC	3 904	6 739	806	614	512	2000–2020
GCC	134 613	609 338	10 058	155 507	132 413	1996–2020
Git	44 081	396 570	11 263	111 715	67 624	2005–2020
Jailhouse	2 717	14 388	354	5 047	3 328	2013–2020
LLVM	183 127	840 956	8 546	416 143	315 821	2002–2020
OpenEmbedded-Core	35 934	142 287	1 640	62 145	35 934	2011–2020
OpenVpn	2 744	20 964	1 163	3 993	2 744	2005–2020
Postgres	36 553	396 393	5 317	28 854	18 343	2002–2020
QEMU	71 588	740 534	9 216	299 483	104 840	2003–2020
U-Boot	60 662	412 668	8 634	143 261	73 509	2000–2020
Wine	156 847	317 498	4 682	241 811	123 168	2001–2020

Intuitively, one would expect that the number of e-mails is somewhat related to the number of commits, as a higher commit frequency may lead to more discussions. However, this is not the case, as we can see from the number of commits and e-mails in projects Coreutils and FLAC, for example. This could be explained through the communication culture of the projects. Some projects may have a culture of only reviewing patches on the mailing lists, while others could also be communicating about other topics on the mailing lists, such as general discussions about the project or announcements (Guzzi et al. 2013). There is also the obvious relation between the number of patches and the number of commits. The number of patches in Table 2 is the total number of patches that could be detected on the mailing list(s) of the projects. This ranges from only 614 patches in FLAC to 416 143 patches in LLVM. Patch-revision sets are a subset of this as it counts all revisions of a patch as just one submission, which is the unit of analysis in our study. Looking at the difference in the numbers of patches and patch-revision sets also indicates how the patch-review is handled in the projects. A large difference, such as in OpenEmbedded-Core, shows that patches get revised often, while a smaller difference, such as in Buildroot, indicates that there are less revisions of patches. In projects that follow a rigorous patch-review process, we would expect the number of patch-revision sets to be larger than the number of commits as this indicates that the commits are created from the sent patches instead of not being reviewed.

Another important metric is the percentage of patch-revision sets of a project that are either answered, accepted, both, or neither. We show this in Figure 3. We can see that most projects have a majority of the patch-revision sets accepted and answered,

which means that the different revisions of a patch made it successfully through the integration process while also being commented by other developers. The percentage of patch-revision sets that are neither accepted nor answered is relatively small (between 1% and 10%) in most of the projects. This is a good sign for a working patch-review process because not many patches get left behind.

Figure 3 holds further information on the integration process of an OSS project. For example, for Wine, 85% of the patch-revision sets are accepted without being answered. This indicates a patch-review process that does not have a lot of communication about patches. It may be that the developers with direct commit rights in the repository, which are usually the developers that do most of the work, commit their changes without review or with review that is not placed on the mailing list. The results for OpenEmbedded-Core (which we call OE-Core in our figures, for brevity) show a similar, yet not so extreme picture as Wine. Another interesting result is the result of LLVM, which indicates that, while a lot of patches receive answers, not a lot of them are accepted. This suggests that the community of LLVM rejects a lot of submissions. We also investigate such peculiarities in the qualitative analysis of RQ_3 in Section 6.3.

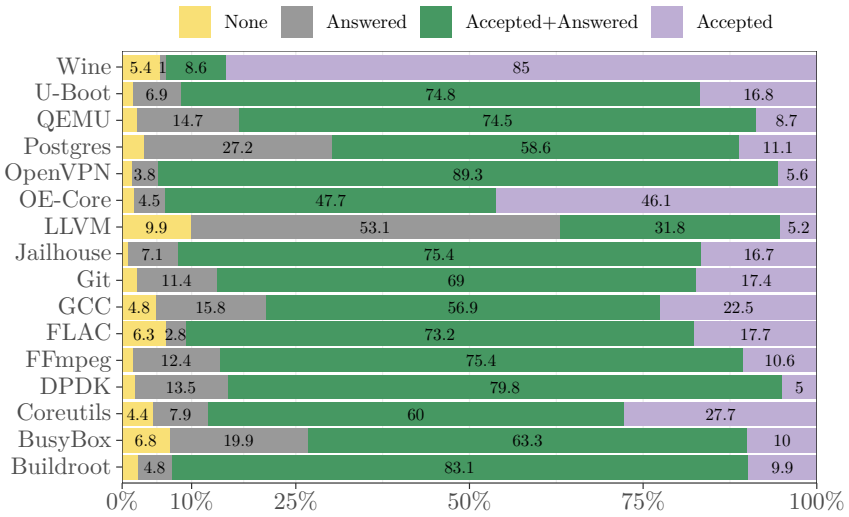

Fig. 3: Percentage of patch-revision sets that were accepted, received an answer, were accepted *and* received an answer, or enjoyed neither.

6 Results

In this section we present the results regarding our three research questions. We start with the results of our quantitative analyses for RQ_1 and RQ_2, and then continue with the results of our qualitative analysis and interviews for RQ_3.

6.1 Probability of Getting Feedback (RQ_1)

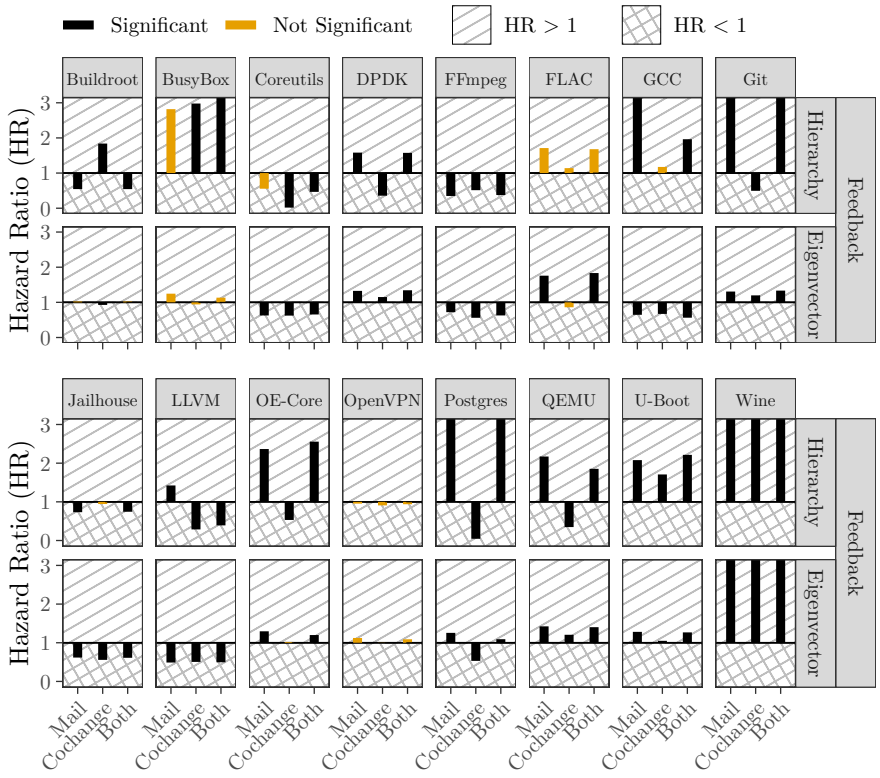

Fig. 4: Hazard ratios (HR) of a developer getting feedback on their patch submission. A hazard ratio above 1 means that a developer with a higher coreness value has a higher chance of getting feedback than a developer with a lower coreness value at any given time. The differently hatched areas are only for better visibility of which hazard ratios are above or below 1.

In our first research question, we ask whether the coreness of developers has an influence on the probability of getting feedback on their patch submissions. The results of our Cox proportional hazard analysis are shown in Figure 4. Overall, the results suggest a mixed picture. There are some projects, namely QEMU, U-Boot, DPDK, Git, Postgres, OpenEmbedded-Core, and Wine, where the results of most configurations support that developers with a higher coreness indeed have a higher chance of getting feedback. In QEMU, for example, the results using 5 out of 6 configurations support that developers with higher coreness values have a higher probability of getting feedback (hazard ratio > 1; counting only significant hazard ratios). The projects, FFmpeg, Jailhouse, LLVM, and Coreutils indicate the opposite (i.e., developers with lower coreness have a higher chance of getting feedback). Similarly to QEMU, Coreutils shows that 5 out of 6 configurations support the opposite of our expectations (hazard ratio < 1; counting only significant hazard ratios).

Finally, there are also some projects that are overall inconsistent and do not show significant differences between developers with higher coreness values and developers with lower ones. These projects are Buildroot, BusyBox, FLAC, GCC, and OpenVPN. In OpenVPN, for example, there is no statistically significant result supporting either our expectation nor the opposite. We discuss the differences between the configurations in more detail in Section 7.3.

Our results show that, while there are projects where the feedback aspect of the integration process shows the expected properties (namely that developers with higher coreness have a higher chance of getting feedback than developers with lower coreness), there are many projects that do not (which is also a plausible result as we have indicated in Section 4.5).

We answer RQ_1 as follows: While developers with higher coreness values have a higher chance of getting feedback on their patch submissions in some projects, there are also projects where this is not the case. In 9 out of the 16 projects, our assumption is not met. This is the majority of projects and therefore we cannot say that our assumption is generally true. However, we also cannot say that our assumption is generally false, as there are still 7 projects that show results in line with our expectations. Of the 9 projects that do not follow our expectations, 4 show that developers with lower coreness values have a higher chance of getting feedback, while the other 5 projects show inconsistent results. Therefore, due to the lack of a clear picture, we cannot draw a general conclusion on the influence of developer coreness on the feedback probability and deem the results to be overall inconclusive.

6.2 Probability of Getting a Patch Accepted (RQ_2)

Our second research question is concerned with whether developers with higher coreness values have a higher chance of getting a patch accepted than developers with lower coreness values. We show the results of our corresponding Cox proportional hazard analysis in Figure 5. Our results draw a mostly consistent and statistically significant picture that developers with higher coreness value (and, thus, probably higher experience in the project) are indeed more likely to get their patches accepted.

However, there are some exceptions. In the projects Buildroot, BusyBox, LLVM, and OpenVPN, we see largely inconsistent results, such as statistically insignificant results and results that jump from a hazard ratio above 1 to a hazard ratio below 1 in the different configurations although BusyBox and OpenVPN still show that half of their results are statistically significant above 1. These are three projects that also show inconsistent results for the analysis of RQ_1, as described in the previous section. There is a strong indication that these findings are linked to each other. One possible reason could be that maintainers (usually developers with high coreness values) do not post their patches onto the mailing list but rather push them directly into the project's VCS. This would then lead to us not being able to find these submissions with our analyses and, thus, not being able to draw consistent conclusions. We explore the actual reasons for these inconsistent results of these three projects during our

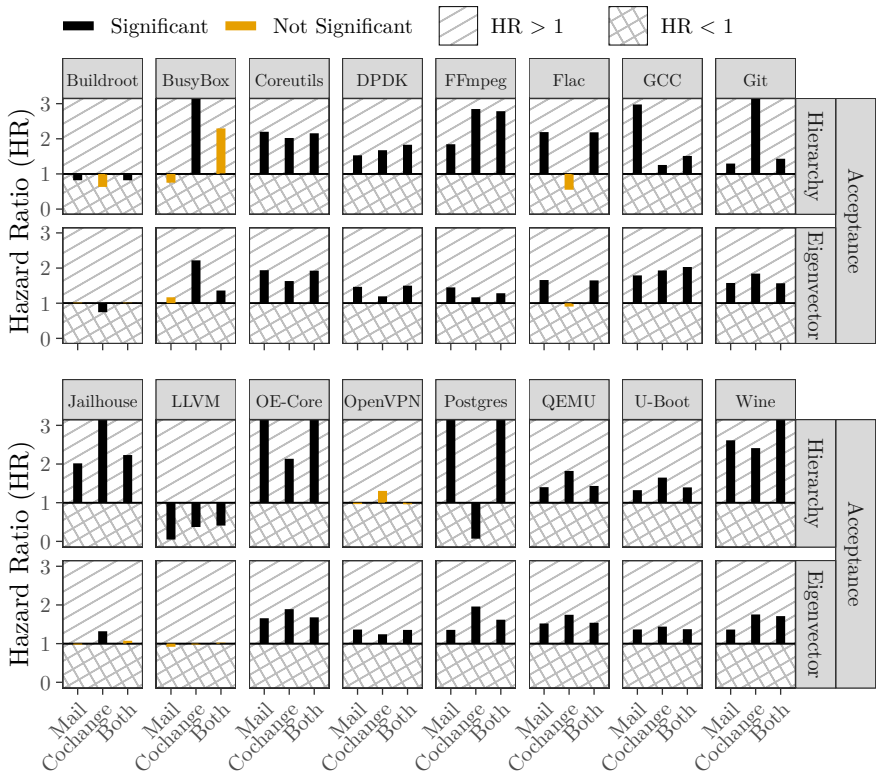

Fig. 5: Hazard ratios (HR) of a patch being accepted under the influence of developer coreness. A hazard ratio above 1 means that a patch by a developer with a higher coreness value has a higher chance of getting accepted at any given time. The differently hatched areas are only for better visibility of which hazard ratios are above or below 1.

qualitative analysis for RQ_3 in the following section. Furthermore, we discuss the differences between the configurations in more detail in Section 7.3.

Looking at the influence of the number of reviewers on the acceptance probability in Figure 6, we see a mostly consistent picture (for all projects except for FLAC) that the fewer reviewers review a patch, the higher the chance of the patch being accepted (hazard ratio < 1; counting only significant hazard ratios). When comparing the influence of the developer coreness on the acceptance probability with the influence of the number of reviewers and without the number of reviewers [13] as a covariate, we see that the results are largely consistent. Therefore, we answer RQ_2 as follows:

While there are some inconsistencies in the results of our analysis of RQ_2, we find that developers with higher coreness values generally have a higher chance of getting their patches accepted in most projects.

[13] We do not present these results as they are mostly the same as the results we present.

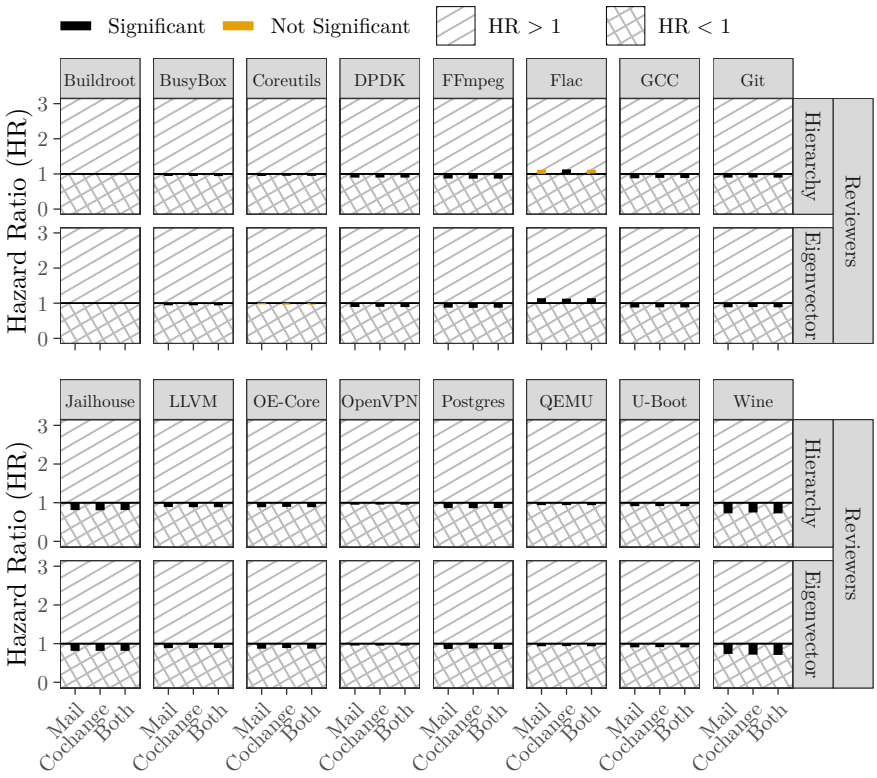

Fig. 6: Hazard ratios (HR) of a patch being accepted under the influence of the number of patch reviewers. A hazard ratio above 1 means that a patch reviewed by more reviewers has a higher chance of getting accepted at any given time. The differently hatched areas are only for better visibility of which hazard ratios are above or below 1.

6.3 Qualitative Analysis of Process-Deviant Projects (RQ_3) and Expert Interviews

So far, we have found that there are some projects that do not show a clear picture with respect to the analysis of the influence of developer coreness on the integration process (i.e., the results regarding RQ_1 and RQ_2 do not show that developers with higher coreness have a significantly higher feedback and acceptance probability). In addition, when looking at the commit coverage that we obtained for our subject projects (see Table 3), we can see that some projects have a very low commit coverage, meaning that a lot of commits in the VCS cannot be mapped to patches on the mailing list. While one could argue that it would be reasonable to exclude the projects with extremely low commit coverage from our analysis, we consider these projects still important to analyze for different reasons. For one, all these projects explicitly state that their patch-review process is based on the mailing list. The low commit coverage, however, indicates that this is not the case in practice. Therefore, these projects are interesting to analyze as they show a discrepancy between the stated patch-review process and the actual patch-review process. Including such projects is important to show the difference

between process-conform and process-deviant projects and to understand why such deviations happen. Secondly, the conclusions of our analysis of RQ_1 and RQ_2 for these projects are still valid for the portion of patches that were sent to the mailing list.

We take both the findings of RQ_1 and RQ_2 into consideration to identify process-deviant projects (i.e., projects whose commit coverage is less than 66.7% or whose results do not reflect our expectations for RQ_2; solely not reflecting our expectations for RQ_1 is not sufficient to be considered to be a process-deviant project, as defined and explained in Section 4.6). Overall, we identify seven projects that fulfill the criteria to be identified as process-deviant, which are indicated in bold in Table 3. To further validate our chosen threshold of 2/3 for the commit coverage, as mentioned in Section 4.6, we also calculated the average commit coverage across all 16 projects, which is 57.28%. This means that the average commit coverage is even lower than our chosen threshold, which supports our choice of the threshold. Moreover, when comparing the commit coverage of the lowest identified process-conform project (COREUTILS with 68.8%) and the highest identified process-deviant project (GCC with 30.2%), we can see a clear distance between these two projects with the upper end being right around our chosen threshold. Note that, while BUILDROOT has a commit coverage of 78.9%, we still identify it as process-deviant because the results of RQ_1 and RQ_2 do not reflect our expectations.

In what follows, we present the results of our qualitative analysis of the process-deviant projects including the answers we received from the expert interviews of these projects. Afterwards, we also present the results of the expert interviews for the projects that we identify as process-conform. In summary, we received answers from experts of 9 projects out of our 16 subject projects.

Projects that we identify as process-deviant:

BUILDROOT: For BUILDROOT, we see in Table 3 that, while the commit coverage is high, the results of RQ_1 and RQ_2 show that developers with higher coreness values neither have a higher probability for getting feedback (see also the inconsistent and partly insignificant results in Figure 4), nor a higher probability of patch acceptance (see also the hazard ratios below 1 in Figure 5). One of the maintainers of BUILDROOT stated that the project has a very strict patch-review process. Anybody that does not have direct commit access to the repository has to submit their patches to the mailing list and the people that do have this access still submit their patches to the mailing list for review. Furthermore, in the early days of the project, there were no such strict guidelines in the project which led to inconsistencies across the code base and, therefore, they adopted the strict integration process. Hence, the inconsistencies in early days of the project and the later adoption of the strict integration process could be reasons for the inconsistent results in RQ_1 and for the unexpected results in RQ_2.

BUSYBOX: We can see in Table 3 that only 5.5% of the commits in the project's VCS could be mapped to patches on the mailing list. We analyzed a sample of the commits that could not be mapped by `PaStA` and manually checked these patches for commonalities. The commits in question have largely been integrated by one maintainer

(whom we will not be naming for privacy reasons). At closer inspection of the patches on the mailing list, we find that there are patches with similar properties. What happens here is that a maintainer takes a patch from the mailing list, rewrites it, and then commits it straight to the VCS. This rewriting hinders `PaStA` to map the commit to the original patch. Furthermore, in many cases, the maintainer does not use the mailing list but pushes patches directly to the VCS. When contacting the maintainer of the project about this unusual practice, we unfortunately did not receive a response.

FFmpeg: For FFmpeg, we are able to map only 28.1% of the commits in the repository to patches on the mailing list. When looking closely at these commits, we find that the developers responsible for most of these commits are or were at some point listed in the `MAINTAINERS` file of the project. This means that these developers have direct push access in the VCS. In particular, 62% of the commits that (even with manual inspection) cannot be traced to patches on the mailing list have been authored and committed by the top-10 contributors with the highest number of commits in the project (all listed in `MAINTAINERS`). That is, the maintainers largely use their direct push access and do not post their patches to the mailing list. A personal communication with a former maintainer of FFmpeg prior to the interviews revealed that maintainers did not post their contributions to the mailing list on a regular basis. In our interview with current maintainers, they told us that this was indeed the case in the past. However, they also mentioned that they have changed this practice and now post most of their patches to the mailing list for review. Triggered by this, we re-evaluated the

Table 3: Overview of the commit coverage (i.e., percentage of commits that can be mapped to patches on the mailing list), whether our expectations for RQ_1 and RQ_2 are fulfilled, and whether we have identified a project as process-deviant or process-conforming. We identify the **bold** projects as process-deviant. × denotes that our expectations regarding one of the research questions are not fulfilled; ✓ denotes the opposite.

Project	Commit Coverage	Expected Result for RQ_1 (feedback)	Expected Result for RQ_2 (patch acceptance)
Buildroot	78.9%	×	×
BusyBox	5.5%	×	✓
Coreutils	68.8%	×	✓
DPDK	90.8%	✓	✓
FFmpeg	28.1%	×	✓
FLAC	8.3%	×	✓
GCC	30.2%	×	✓
Git	81.1%	✓	✓
Jailhouse	89.1%	×	✓
LLVM	8.1%	×	×
OpenEmbedded-Core	91.9%	✓	✓
OpenVPN	69.3%	×	✓
Postgres	16.2%	✓	✓
QEMU	89.1%	✓	✓
U-Boot	92.1%	✓	✓
Wine	69.1%	✓	✓

commit coverage for just the last two years in our data and found that the commit coverage indeed improved to 77.3% for the years 2019 and 2020. This shows that the project is actively working on improving the transparency of the integration process.

FLAC: Similarly to BusyBox, the project FLAC is also mainly developed by very few but very active developers. As we can see in Table 3, we can only map 8.3% of the commits to patches on the mailing list. When inspecting these commits more closely, we found that they are mostly related to these few maintainers. These maintainers do not use the mailing list for their patch submissions. Since FLAC is a rather small project, it is reasonable to assume that these few maintainers might communicate through other channels than the official mailing list. The current maintainer of FLAC confirmed that the project has a small number of active developers (only one really active maintainer for long stretches of time). These few developers do not post their patches before integrating them, because there is no real community that could review the changes. Nevertheless, this makes the integration process intransparent and, in the worst case, this can lead to a lack of proper quality control.

GCC: This project has a commit coverage of 30.2%. When looking closely at why this is the case, we found that this is mainly due to technical difficulties. Specifically, we find that a lot of the unmapped commits are either from the previous SVN repository (an older VCS where the commit history is not easily transferred to Git) or came into the project via some sub-projects (included in the project as libraries) that we do not track. In particular, on occasion, the project copies snapshots from external dependencies into their repository via a single commit that is comprised of many individual changes to the external dependencies that have appeared as individual e-mails on the mailing list, which makes mapping the resulting large commit to individual patches on the mailing list infeasible.

Independent of these technical difficulties, it appears that there is no concrete guideline in GCC on how patches have to be integrated through one central mailing list. When contacting some core contributors of the project, we engaged in a discussion about the integration process of the project with them. It seems that we are not able to map a lot of the commits to patches on the mailing list because the project does not enforce formatting (e.g., `format-patch` in Git) for the patch submissions. This means that developers can submit their patches in any way they like, which makes it hard to track these patches. Furthermore, they told us that it is difficult to enforce new rules because some long-standing contributors are used to their process. This does not necessarily mean that the project has a bad integration process, but it is hard to track the process with our methods and could therefore be potentially intransparent for new contributors.

LLVM: For LLVM, there are similar problems as for GCC leading to a commit coverage of 8.1%. Also, the results of RQ_2 are inconsistent and do not reflect our expectations. When looking at the commits that could not be mapped to patches on the mailing list, we found that a lot of them are related to the LLVM sub-projects (e.g., Clang, LLDB, etc.) that are included in the main LLVM repository. The community

of this project changed the contribution system several times (e.g., from a mailing list to Phabricator[14]), which limits our ability to map the commits that did not get reviewed on the mailing list. Moreover, there are some security-related discussions in this project that are not publicly available and thus cannot be mapped to commits in the project. These inconsistencies in the integration process could also explain the inconsistent results for RQ_2. When contacting core contributors of the project, we did not receive a response.

Postgres: Postgres is considered process-deviant because of the low commit coverage of 16.2%. During our manual, qualitative analysis of the project's integration process, we found that a substantial number of the unmapped commits are actually linked to conversations on their mailing lists (e.g., via their CommitFest App, which is a tool to track the progress of developers' patch submissions[15]). Consequently, we investigated why `PaStA` was not able to detect some of these patches on the mailing list and found several process-related or technical reasons for that: (1) In some cases, the patches on the mailing list were too different from the commits in which they eventually resulted, which could indicate that the developers make significant rewrites of some patches that are not posted to the mailing list again. (2) In other cases, we figured out that the developers modified the traditional patch-review process (e.g., the patch is reviewed and discussed on the mailing list after the patch has already been integrated into the VCS) or did not adhere to it at all (e.g., a patch is integrated without any review). (3) We found a substantial number of e-mails that contain multiple patches as separate attachments. `PaStA` is not able to handle separate patches that are contained in a single e-mail, as this would violate the uniqueness constraint of identifying a patch by the corresponding message id. In such cases, `PaStA` considers only the first attachment that contains a patch and ignores the remaining attachments. (4) Finally, we also identified other technical difficulties in e-mail formats that let standard e-mail-processing tools fail on processing the e-mail content (e.g., special forms of delimiters for GPG signatures). In summary, there are process-related reasons (i.e., deviations from the usual patch-review process) as well as technical difficulties (i.e., limitations of our patch-detection methodology) that both contributed to the low commit coverage in our analysis, which is why we identify Postgres as a process-deviant project. When contacting core contributors of the project, we did not receive a response.

Projects that we consider to be process-conforming:

OpenVPN: While OpenVPN is not considered a process-deviant project because only the results of RQ_1 regarding received feedback do not fulfill our expectations, and we have already provided plausible reasons for such a result in Section 4.5, we nonetheless want to highlight this project since the results of RQ_1 are still somewhat unique in that they show no statistical significance, at all. The maintainers of the project told us that OpenVPN is an old project that has gone through many major changes to its development process. Examples include changing the VCS and a changing

[14] https://www.phacility.com/phabricator/
[15] https://wiki.postgresql.org/wiki/CommitFest_App/

developer base. Furthermore, they explained that the project had to evolve towards its current integration process over time going from a 'one-man show' to a team effort. We again looked at the last two years our data for OpenVPN (i.e., 2019–2020) and found that, in these last two years, 99% of the commits can be mapped to patches on the mailing list. This shows that the project has made significant progress in improving the transparency of the integration process.

Wine: While Wine is also not considered a process-deviant project regarding our definition in Section 4.6, we still investigated the particularly high hazard-ratio values for the analysis of RQ_1 (for which we observe hazard ratios of over 100 for Wine as opposed to hazard ratios below 10 for all the other projects). This is neither an inconsistency nor an insignificant result, but an interesting finding. The reason is that there is one developer (who we will not name for privacy reasons) who was extremely active on the mailing list. This developer was involved with almost all patch reviews and discussions on the mailing list and also received feedback by the community on their submissions almost every time. Thus, the hazard ratio takes on huge values. Excluding this developer from the analysis, the results for Wine are still consistent and in line with our expectations, but in a much lower range of the hazard-ratio values. In addition, as this developer left the project sometime in 2019, we updated the data for Wine to also include the year 2021 to have enough data afterwards to be able to investigate how the leaving of this developer has impacted the project's feedback process. We repeated our analysis for the feedback probability (RQ_1) once with all data from the years 2017 to 2018 and once with the data from 2020 to 2021 and show the corresponding results in Figure 7. We can see that, if we look at the two years after this developer had already left the project, the hazard ratios take on much smaller values than before. This shows that the abandonment of a central developer in a project can have an important impact on the feedback process in the project. When contacting core contributors of the project, we did not receive a response.

DPDK: The maintainers of DPDK told us that the project has a very strict integration process meaning that all patches must be posted to the mailing list for review, which is in line with our results. They emphasize that this increases the number of reviews a patch gets and thus the quality of the code base.

Jailhouse: As one of the authors of this article is a core contributor to Jailhouse, we are able to obtain expert insight into this project without contacting the maintainers of the project. He details that the project has a strict integration process where most patches have to be posted to the mailing list for review, which is in line with our results.

OpenEmbedded-Core: The maintainers told us that they enforce a strict integration process in this project, which is in line with our results. They further elaborate that this is necessary because the scale of the project is too large to be managed without a strict integration process. They state that this helps their project to maintain a high-quality code base and that the community is happy with the process.

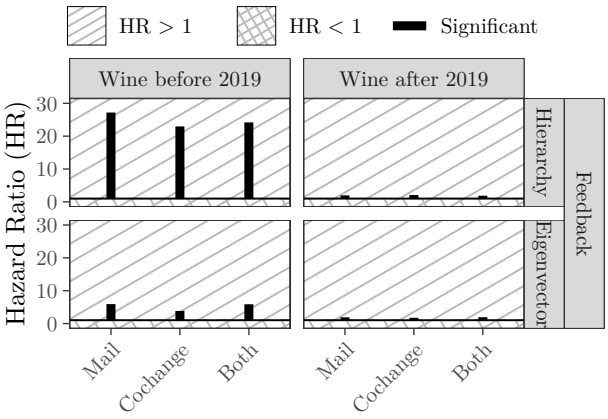

Fig. 7: Hazard ratios (HR) of a developer getting feedback on their patch submission for the project Wine in the two years before 2019 and after 2019. The hazard ratios of Wine after 2019 take on much more moderate values, since the very active developer that caused the high values was not involved in the project anymore. The differently hatched areas are only for better visibility of which hazard ratios are above or below 1.

U-Boot: We received detailed answers to our questions from the contacted maintainers of U-Boot. They told us that they indeed have a strict integration process except for some security-critical patches that get discussed in a smaller circle of developers. But even these discussions are later posted to the mailing list. Further, they named three key reasons they think a strict integration process is important. These reasons are (1) that the feedback received from the community is valuable for improving code quality, (2) that the development process is archived through the discussions on the mailing-list, and (3) that the developers are familiar with the current process which can help the flow of development.

Coreutils, Git, and QEMU: For these remaining three projects, we did not receive any responses to our inquiries.

Overall, we answer RQ_3 as follows: Unexpected results in our analyses of RQ_1 and RQ_2 often coincide with procedural problems in the integration process. Thus, we conclude that an unexpected result in a quantitative analysis of the integration process can be an indicator of procedural problems. Our expert interviews indicate that the developers are often aware of these problems and are actively working on improving these inconsistencies in the integration process if given the chance.

7 Discussion

In this section, we discuss our results and their implications.

7.1 Probability of Getting Feedback on Submissions (RQ_1)

Our first research question addresses whether a developer's coreness has an influence on their probability of getting feedback on their submissions. In line with the results of Bosu and Carver (2014), we find that a higher ranking in the coreness scale does seem to have an influence on the probability of getting feedback. Although there are mixed results shown in Figure 4, we find that in the projects in which the process of submitting a patch via the mailing list is followed (e.g., QEMU and U-Boot), developers with a higher coreness value do have a higher probability of getting feedback on their code submissions.

As described in Section 4.5, this does not necessarily indicate procedural problems as project communities might be more inclined to generate quicker feedback on code submissions by less experienced, newer developers to try and retain them for their projects. Overall, when looking only at the projects that we consider as process-conform in our analysis of RQ_3, we see that developers with higher coreness values do seem to have a higher chance of getting feedback than developers with lower coreness scores. This means that this observation only holds for projects that strictly adhere to a systematic integration process.

7.2 Probability of Patch Acceptance (RQ_2)

Our second research question is concerned with whether developers with a higher coreness value have a higher probability of getting their patches accepted than developers with a lower coreness value. Again, similar to Bosu and Carver (2014), we find that this seems to be the case. As is visible in Figure 5, most projects show significant hazard ratios above 1 across all configurations of network type and coreness metric.

There are, however, some projects that show inconclusive results. We again hypothesize that this can be attributed to the inconsistencies in process conformance. One such project is BusyBox. There, we can see that the results are not consistent but show a mix of statistically significant and insignificant results. In contrast, projects with proper process conformance, such as U-Boot and Wine, show consistent results, that is, all results are statistically significant and consistently yield a hazard ratio larger than 1.

When looking at the hazard ratios of the influence of the number reviewers on the probability of patch acceptance in Figure 6, we find that, in most projects, fewer reviewers lead to a higher probability of patch acceptance (although only slightly).

7.3 Cross-Configuration Insights

As we have described in Section 4.3, we use six different configurations for our analyses of feedback (RQ_1) and acceptance (RQ_2) to ensure the robustness of our results. In this section, we discuss the insights we can gain from looking at the results across these different configurations. For the majority of the projects and for both research questions, we find that the results of the combined network are very similar to the results of the mail network. This is visible in the results for feedback in Figure 4 as well as the results for acceptance in Figure 5, where the results of the combined network and the mail network are almost identical for most projects. To be precise, in 13 out

of 16 projects in the results for feedback and in 14 out of 16 projects in the results for acceptance, the results of the combined network are aligned with the results of the mail network. Overall, this makes sense as the mail network is on average more dense and larger than the cochange network, which means that the mail network has a stronger influence on the combined network than the cochange network. Therefore, with all developers included in the combined network, a majority of the edges in this network are still edges from the mail network.

The mail and combined network also show a slightly more consistent picture across the analyses of feedback and acceptance than the cochange network. Especially for feedback this makes sense as the mail network captures the activities that are being analyzed in this research question, that is, the feedback on patches that is given on the mailing list. For the analysis of acceptance, this difference, while still present, is smaller than for the analysis of feedback. However, since the overall picture of the results of acceptance is more consistent than for feedback, the differences between the different network types are less pronounced as for the analysis of feedback.

The difference between the two coreness metrics is not as strong as the difference between the network types. This serves as an additional robustness check for our results as the results are not dependent on the coreness metric used (Bock et al. 2023). The most notable difference between the two coreness metrics is that the hazard ratios of the configurations involving hierarchy centrality are usually higher than the hazard ratios of configurations that use eigenvector centrality. This can be explained by the very nature of the metric. For hierarchy centrality, there are usually fewer developers with higher coreness values (i.e., few developers with very high coreness and many developers with relatively low coreness), and if these developers have a higher probability of getting feedback or getting their patches accepted, this will lead to higher hazard ratios than for eigenvector centrality, which usually has a more even distribution of coreness values across the developers.

In summary, the results across the different configurations, while not identical, are mostly consistent and the inconsistencies stem mostly from the different aspects of the socio-technical nature of the OSS projects that are captured by the different network types and coreness metrics. This also shows that there is not one single best way to capture the organizational hierarchy of OSS projects and that it is important to look at different aspects to get a more complete picture of the processes in these projects.

7.4 Qualitative Analysis (RQ_3) and Expert Interviews

With our third research question, we ask whether the unexpected results we find in our analyses of RQ_1 and RQ_2 indicate procedural problems in the integration process of these projects. We found that there are indeed some projects that show procedural inconsistencies. These are mostly related to developers with direct push access to the project's VCS, not using the mailing list for patch reviews. Since the integration process is the quality control of OSS projects, this can lead to severe problems as the quality of the source code can not be guaranteed anymore. Our expert interviews with a subset of the process-deviant projects reveal that the developers are aware of such problems and some are actively working on improving these inconsistencies. In other cases, the modalities of the project simply do not allow for such a strict integration

process because the community is too small or the project is too specialized. Although developers that have direct push access to a project usually have substantial experience with the project and should be able to produce quite high-quality code, there can always be oversights or errors when coding and these might not be detected in time without a proper review. This can then lead to bugs and even security concerns as indicated in the literature (Cairo et al. 2018).

In our qualitative analysis, we also found that each project has its own peculiarities with respect to its developer community and its integration process. For example, some projects use modified patch-review processes and patch-submission guidelines (e.g., POSTGRES), while other projects have changed their processes during the project's evolution (e.g., BUILDROOT or OPENVPN). Thus, investigating whether a project adheres to its contribution and integration processes is difficult to judge, given the evolution of projects and their communities, and also taking technical difficulties into account. Consequently, when identifying process-conformance, there will always be projects that are on the border between being considered to be an process-deviant or process-conformant, no matter which characteristics and thresholds are used to determine this. For instance, OPENVPN and BUILDROOT show similar characteristics (both acknowledge that they have improved their processes over time) and a similar commit coverage. However, as BUILDROOT is one of only two projects (the other one being LLVM) that does not follow our expectations regarding patch acceptance, we consider it to be an process-deviant. On the other hand, OPENVPN is the only project that does not show any significant results for RQ_1, meaning that a developer's coreness does not have a significant influence on the probability to receive feedback. But we consider it to be a process-conformant project, as there are plausible reasons why receiving feedback might be independent of a developer's coreness. This shows the flexibility of our multi-method methodology and allows for individual project peculiarities and for cases in which no significant differences can be observed, which is a major advantage of our method. That is, a general, quantitative mechanism to identify process-conformance is not useful without qualitatively investigating the corner cases. Unexpected results in quantitative analyses can, however, be an indicator for potential procedural problems as we have seen in some cases in our qualitative analysis (e.g., FFMPEG or BUSYBOX). Also, the qualitative analysis we did on the projects that inhibit unexpected results in our quantitative analyses gave us important insights for understanding the socio-technical collaboration processes in OSS projects.

All in all, we received answers from 9 of our 16 subject projects when asking about their integration process. This indicates that there is a general interest in the integration process of OSS projects. Most developers told us that they think a rigorous integration process is important and necessary to ensure the quality of the source code and stay transparent with the development community. As one developer put it when asked whether they think it is important to enforce a strict integration process:

> "[...] *If I take a look at it from the other side, i.e. pose a question—why not apply my own patches and send it to upstream without posting them to the lists—that is a question I have hard time answering. To me, this is a similar question to—why do you not litter in public—or—why do you not throw random pieces of trash on*

the ground outside—I don't know the exact answer to that, simply because it is not good behavior, and if people did that, the environment around us would look horrible. [...]"

Responses such as this one show that the communities of OSS projects are aware of the importance of the quality control in their projects.

7.5 Implications of our Findings

Our findings address a key element of OSS development: the patch-integration process. This is the quality control mechanism of OSS projects and ensures that the standards of a project are maintained. Without a systematic integration process, security flaws and other problems could not be prevented effectively.

Another key ingredient of OSS development are the developers. We found that developers with a higher coreness value do indeed have a higher probability of finishing the integration process successfully and, in many cases, also of receiving feedback from the community. As there is evidence in literature that successful interactions among developers reduce the chance of developer turnover (Bosu and Carver 2014; Jiang et al. 2013), this is an important finding. This finding shows that the community implicitly is more concerned with keeping active and well-connected developers in the project. This can have many reasons: Patch submissions by experienced developers are of higher quality than submissions from less experienced developers, and thus these developers could have a higher chance of getting these patches accepted. Furthermore, more experienced developers should already have working relationships with the maintainers of OSS projects, meaning that they know exactly whom to contact when looking for a review. This leads to these developers' submissions having a higher probability of getting feedback faster from the community than less experienced ones.

Nevertheless, our data show that less experienced developers have a harder time with the integration process. This can be demotivating and, as a result, negatively influence the influx of new developers to the community, limiting the pool of potential contributors. This should be analyzed by the respective communities to have the ability to steer their integration process in a different direction.

In some projects, we found inconsistencies with the conformance of the integration process. If a group of developers (especially the core group) do not follow the project's contribution guidelines, this can be demotivating for the whole community. The former maintainer of FFmpeg that we contacted about the inconsistencies in their integration process (see Section 6.3) told us that the lack of coherence to the rules of the integration process was a main reason that the developer ultimately left the project. This confirms that non-coherence with the integration process can have a large impact on the developer base and should be reviewed by the communities. Nevertheless, when contacting current maintainers of FFmpeg, we find that they are actively working on these problems. This shows that these problems are important to the developer communities and that some are actively working on improving their processes. These findings also show that our methodology could serve as a sort of warning signal for project communities to identify potential procedural problems in their integration process and to take action before these problems become a threat to the quality of the

project and its longevity rather than relying on individual developers to raise concerns, which can come at a high personal cost as shown by the example of FFmpeg.

Taken together, our findings indicate that the patch-review process is not merely a technical process but rather a loose social contract between the developers of an OSS project. If this contract is not upheld by the developers on top of the organizational hierarchy, this can lead to a lack of trust in the process itself, which is the quality control of OSS projects. Our methodology offers a practical means of monitoring the conformance to the patch-review process and can be used by project communities to identify potential procedural problems in their integration process and to take action before these problems become a threat for the project and its community.

7.6 Perspectives

The methodology and findings presented in this paper have practical implications that extend beyond the specific projects and setting we analyze. We discuss two directions here: the practical applicability of our methodology, and the potential transferability of our methodology and findings to other types of software projects.

As we described in Section 7.5, our methodology can be used by project communities to look at the conformance to their patch-review process and to identify potential procedural problems in their patch-review process. Practically, this means that project communities could, in principle, apply our methodology periodically to monitor their process conformance. Deviations from the expected coreness–feedback and coreness–acceptance relationships could serve as an early warning signal for emerging procedural problems. In the same vein, companies that contribute to or depend on OSS projects could use this methodology to assess the transparency and rigor of a project's patch-review process before committing resources to it. As our expert interviews suggest, project communities are generally receptive to such findings and willing to act on them, which is an encouraging sign for the practical value of tools built on this methodology.

To discuss the transferability of our methodology and findings to other types of software projects (e.g., commercial closed-source projects, or projects that use issue trackers instead of mailing lists), we necessarily move beyond the scope of our empirical data and into reasoned conjecture. In the design of our study, we consciously chose to analyze OSS projects that use mailing lists for their patch-review process and that follow the RTC model (see Section 2.1). This is because these projects, while using a somewhat dated communication channel in a time where many projects have moved to more modern contribution and communication channels such as GitHub (Zhu et al. 2016), still represent a large part of the OSS ecosystem (Mannan et al. 2020; Ramsauer et al. 2019; Käfer et al. 2018) and are especially prevalent in the domain of system software, which is of high importance for our society. The transferability of our methodology to projects that use issue trackers for their patch-review process is, in principle, possible as the underlying process is similar to mailing lists (Zhu et al. 2016). The mail network would need to be replaced with an issue interaction network, which could be constructed from the issue-tracker data. In addition it is also not trivial to map the patches within an issue tracker to commits in the VCS. If a project, for example, uses a pull request-based process, the pull requests would need to be mapped to commits in the VCS, which can be technically challenging as there

is not a one-to-one mapping between pull requests and commits. This could further be complicated by projects that use squash merging as then the commits that are integrated from the pull request are not the same as the commits that are in the pull request itself. Overall the process is largely the same, but the technical implementation of the data extraction and processing would need to be heavily adapted to the different data sources. Whether or not the results would be similar is an open question that shall be addressed in future work, but we suspect that similar patterns could be found. Although one could argue that it is easier to create a pull request on GitHub than to format and send a patch to a mailing list (Zhu et al. 2016), developers with push privileges could still bypass the review. Similarly, social coding platforms could help with a better overview of currently open discussions as it is easier to track the status of a pull request than the status of a patch on a mailing list, but this does not necessarily mean that pull requests would always receive feedback. All in all, we suspect that the results for issue tracker projects would be largely similar to our results.

The transferability of our methodology and findings to commercial closed-source projects is more difficult to assess. The main difference for such projects is that there is usually a more formal organizational structure and hierarchy in place, which could lead to a more formalized patch-review process. This could lead to a more consistent conformance to the patch-review process and thus to more consistent results across projects. However, this does not mean that the process is always strictly followed and there can be exceptions. Nevertheless, the general idea of our methodology, that is, using network metrics to monitor process conformance and to identify potential procedural problems, is still applicable in such a setting.

8 Threats to Validity

As with any data-driven empirical analysis of OSS projects, our study is prone to threats to validity. In the following, we discuss internal and external threats and how we address them.

8.1 External Validity

Analyzing 16 OSS projects threatens external validity in that our findings are not applicable across the entire OSS landscape. To address this threat, we chose a diverse set of projects that differ in size, age, and domain. We also chose projects that are well-established and long-lasting in most cases. With this diversity, we hope to mitigate the threat of our results not being generalizable to other OSS projects, especially since our analyses are concerned with the patch-review process, which is, in its core, similar in most OSS projects (Rigby et al. 2014; Asundi and Jayant 2007).

Another threat is that all of our subject projects use mailing lists as their main tool for patch submissions. Our results could be different if we chose to analyze projects with another means of collaboration, such as issue trackers. We suspect that this should not be an issue, as these projects use essentially a similar kind of integration process.

8.2 Internal Validity

A threat to internal validity arises from our choice to use social and technical developer networks as a means to infer community structures. This could be problematic, as the community structures could be different in reality. Albeit, since the usage of social and technical developer networks is a well-established method in software-engineering research (Bosu and Carver 2014; Crowston et al. 2006; Joblin et al. 2017a; Bock et al. 2023) where strong evidence of the method's accuracy exists (Meneely and Williams 2011), we can accept this.

Another threat to internal validity is the data set we use for our analyses. We use publicly accessible VCS repositories to mine all commit information from. This can be done via a standardized method and therefore does not pose a risk for our commit data. The mailing-list data were obtained from well-established project-specific archives that allow us to mine all the information via an API. A further threat is that there might be mailing lists we have missed. To mitigate this risk, for all of our subject projects, we did a rigorous search for all possible mailing lists that are mainly used to send patches and are publicly accessible. So, we are confident that we only miss outliers and cover all relevant mailing lists for patch submission (see Table 1).

Furthermore, the patch-detection procedure threatens the validity of our study. As we describe in A, we have taken considerable effort towards reliably detecting patches that do not follow standardized formatting. Through our qualitative analysis of the integration process, we found multiple pitfalls when detecting patches on mailing lists. However, we already addressed a lot of them in our analyses. Most importantly, we use `PaStA`, an established tool (Ramsauer et al. 2019) to map the detected patches to the respective commits in the projects' VCS repositories. While this tool has some limitations, such as not being able to detect more than one patch in the attachments of an e-mail, its high Fowlkes-Mallows index of 0.988 in mapping patches to commits (see Ramsauer et al. (2019)) and the extensive effort we put into cleaning the data (see Appendix A) gives a high enough confidence in the reliability of our patch-detection procedure. Another threat related to the mailing lists is that feedback on submissions might be given via other channels than the mailing list. Such feedback is not traceable for us but we suspect that this happens in very rare cases, where the author of the patch and the reviewer are already communicating via other channels (e.g., chat apps). This should not be the case for most submitted patches, given that the mailing lists of our subject projects are the dedicated and established channel for patch submission and review.

In a similar vein, the detection of replies to a patch on the mailing list is a potential threat. As described in Section 4.2, we use a heuristic to identify replies to a patch on the mailing list. We look for e-mails in the same thread that are sent after the patch has been sent and that are sent by another developer than the patch submitter. If there is no such e-mail, we assume that there is no feedback for this patch. To mitigate this threat, we have performed a sanity check by sampling 10 patch-answer pairs from each project (160 total) and manually checked whether these are actual answers, which we detail in Section 4.2 Another potential drawback to our heuristic is that there might be off-list feedback on patches (e.g., via private communication or other communication channels). This is not something we would have been able to

detect. However looking at Figure 3, we can see that a majority (over 75% on average) of the patch-revision sets receive feedback on the mailing list.

While we choose to focus on the influence of developer coreness, there are other factors that could influence the probability of receiving feedback on a patch or the acceptance of a patch. For example, Weißgerber et al. (Weißgerber et al. 2008) find that the size of a patch influences the probability of patch acceptance in that smaller patches are more likely to be accepted. To address this threat, we have extracted the sizes (sum of added and deleted lines) of the analyzed patches of three of our subject projects (BUILDROOT, JAILHOUSE, and GIT[16]) and included the patch size as a covariate in our Cox proportional hazard models. We find that the patch size has a negligible influence on the probability of receiving feedback on a patch or the acceptance of a patch. Moreover, the results of the influence of developer coreness do not change in both cases. We have also added the number of reviewers of a patch as a covariate in the analysis of RQ_2 (see Section 6.2). While we cannot rule out that there are other factors that influence the patch-review process, we are confident that developer coreness is a major factor that influences the process.

Additionally, our choice of the 2/3 threshold for identifying process-deviant projects is a potential threat to validity. As described in Section 4.6, we chose this threshold because in a previous study of the LINUX KERNEL (Ramsauer et al. 2020) the authors found that about 98% of the commits in the VCS could be mapped to patches on the mailing list. This is exceptionally high and we wanted to allow for a more lenient threshold to account for the fact that other projects might not enforce the patch-review process as rigorously as the LINUX KERNEL does. Additionally, we compared the chosen threshold against the average commit coverage of our subject projects, which is 57.28% and therefore clearly below our threshold, which gives us further confidence in our choice. Lastly, we also observe a considerable gap between the project with the lowest commit coverage above the threshold and the project with the highest commit coverage below the threshold, which also supports our choice of the threshold (see Section 6.3). Taken together, we are confident that our choice of the threshold is reasonable and does not threaten the validity of our results.

Next, our semi-structured expert interviews pose a possible threat to validity for multiple reasons. One reason is the selection of experts. We cannot entirely be certain that we chose the appropriate experts for each of our subject projects. We did, however, research the projects in detail to determine which developers would be most suitable for our interviews. We consider suitable developers to be those who have been active in the project for an extended period of time, are still active, and either have high contribution numbers or even get identified as maintainers by the project communities themselves. We assume that such developers have enough knowledge of the project and the processes used in the projects.

Finally, there is the threat of the posed questions. We took great care when formulating the questions we sent to the chosen experts. In our opinion, these questions reflect the purpose we designed them for, but some experts might have interpreted our questions differently, though. Lastly, while we received answers from more than 50%

[16]We have chosen these three as they follow standardized patch formats on their mailing lists which makes it possible to extract the patch sizes. These three projects also cover process-deviant projects, process-conform projects, and projects of different sizes.

of our subject projects, we cannot be certain that the received answers reflect the opinions of the remaining projects. But since the answers we received were similar in their meaning, we suspect that they are representative.

9 Conclusion

The code integration process is one of the most important steps in the OSS development process as it serves as the quality gate for source code and functionality. Understanding the factors that influence this process is crucial for improving software quality and detecting procedural problems in the code integration process. Therefore, in this study, we investigated whether a developer's coreness has an influence on their success in the code integration process of OSS projects. We use developer coreness as a proxy-measure for a developers' position in the organizational hierarchy in our subject projects. Corroborating previous work (Bosu and Carver 2014), we found that, in many projects, a higher coreness value appears to positively influence the probability for a developer to receive feedback on their patch submissions, and, in general, to ultimately get their patches accepted and integrated. We triangulated these findings by using a comprehensive quantitative analysis followed by an in-depth qualitative analysis of the results, including expert interviews with developers from the subject projects. A key insight of our study is that the conjecture that developers with higher coreness have a higher probability to receive feedback and to have their patches accepted usually only holds for projects where the community follows a systematic process of sending all new patches onto the mailing list for review. In projects where (some) developers are less strict when keeping up with the submission guidelines, our results are less consistent and pronounced. We even find that in these situations there are possible procedural problems that can lead to issues with the quality and security of the software through an in-depth qualitative analysis. This shows that our methodology can be used to find indications of procedural problems in OSS projects, which is a prerequisite for solving these problems. By means of semi-structured expert interviews, we confirm that the project communities are interested in such findings and are actively working on fixing procedural problems in the integration process.

Possible follow-up research includes the investigation of whether a lack of process conformance by the top 10% of the developers in the coreness scale does indeed have an influence on the rest of the community and whether this increases developer turnover. Also, our study could be extended by determining whether simple analyses of the development processes in OSS projects can be used as an early-warning mechanism for possible problems in the project's development process.

Appendix A Fixing Different Patch Formatting Issues

Unlike for patches submitted to pull-request-based systems such as *GitHub*, patches submitted to mailing lists do not necessarily follow a standard format. While identifying and mapping patches using `PaStA`, we had to deal with various patch-format inconsistencies, which we identified during our analysis.

To be able to judge the completeness of our patch-identification approach on our subject projects, and also to judge the conformance to the patch-submission process

in these projects, we investigate for each project how many of the commits in the VCS we identify as patches on the mailing lists using `PaStA`. Therefore, we compute the *commit coverage*, which is the ratio of commits in the VCS that can be mapped to patches on the mailing list. The base version of `PaStA` did not provide a satisfactory coverage of patches for some of our subject projects (e.g., only 16% for project GCC), so we had to improve the patch-detection process in `PaStA` to be capable of identifying a higher percentage of patches (and, thus, possible commits) on the mailing lists for various projects and tens of years.[17] This motivated us to further investigate commits that could not be mapped to a patch, yet, to understand why the commit coverage of `PaStA` was surprisingly low, and to improve our mapping approach in order to increase the commit coverage. Note that a low commit coverage prevents us from drawing conclusions about feedback and patch acceptance, as most of the patches are not covered by our data then, indicating that either our data are incomplete or the developers did not comply with the patch-review process.

By means of qualitative inspections, we found multiple issues regarding patch formats, which we solved within the patch-detection functionality in `PaStA`. Many projects used various, partially non-standardized formats for patch submission: Patches usually consist of a *commit message* (textually describing the patch) and a *diff* (containing the actual changes in terms of added lines and deleted lines of the changed files) (MacKenzie et al. 2003; Ramsauer et al. 2019). After our investigations and improvements, `PaStA` is able to detect patches that contain both, commit message and diff, as well as patches that just contain a diff. Patches can be either the content of an e-mail (which is the normal case if patches are generated via `git-send-email`[18], for instance) or can be added as an attachment to an e-mail. When searching for patches in attachments, we consider all file extensions, since we found patches in files that have an expected file extension (e.g., ".patch", ".diff", etc.) but also in files with arbitrary file extensions (e.g., ".txt", etc.). However, if there are multiple attachments in one e-mail, we only consider the first attachment containing a patch and neglect the remaining attachments, as our processing procedure restricts e-mails (identified by their message id) to be considered as a single patch. Nevertheless, we do not consider this as a substantial threat since it occurs only rarely that an e-mail contains multiple attachments with patches (except for project PostgreSQL, which we discuss in detail in Section 6.3).

When parsing diffs in `PaStA` (which is necessary to be able to map commits to patches on the mailing list), we empirically observed a variation in diff formats that have been used in patches: On the one hand, diffs in *unified mode* are common; it is the standard mode for patches generated via Git[19]. On the other hand, there have also been many patches containing diffs in *context mode*[20] (which was considered the standard format prior to unified diffs (MacKenzie et al. 2003)), especially in projects PostgreSQL and GCC. As context-mode diffs can automatically be converted into unified-mode diffs, we extended `PaStA` to also map patches that use unified-mode diffs

[17] In Table 3, we provide the final commit coverage for each of the projects after our improvements to the patch-detection process.
[18] See https://git-scm.com/docs/git-send-email (accessed at 2022-07-22)
[19] See https://git-scm.com/docs/git-diff (accessed at 2022-07-22)
[20] See https://www.geeksforgeeks.org/diff-command-linux-examples/ (accessed at 2022-07-22)

to commits in the VCS. With this ability, the commit coverage for project GCC, for example, was increased from 16% to roughly 30%, which we consider a substantial and important improvement.

In addition to that, in our qualitative inspections, we also identified deviations from the standard diff format in the first line of a diff: The first line of a diff usually starts with `diff --git`, but there are also diffs where this line was missing. Also, differences in the filename patterns of a diff occurred: Usually, in a diff, the version of the file before the change is named `a/filename` and the version of the file after the change is named `b/filename` (with `filename` being a placeholder for the actual name of the changed file). However, there are also patches using `old/filename` and `new/filename`, instead, or any arbitrary alpha-numeric strings in front of `/filename`. To improve our commit coverage, we improved the diff parsers in `PaStA` to deal with these and various other diff formats.

In summary, we made huge efforts to manually investigate patch formats in detail and improved our automatic patch identification functionality accordingly.[21] As a consequence, our extensions to `PaStA` increased the number of identified patches on the mailing lists significantly (e.g., for GCC, the base version of `PaStA` identified about 100 900 patches in e-mails on the mailing lists, whereas our extended version of `PaStA` identifies 155 204 patches on the mailing lists for the same investigated time period, which is an increase by 54% for this project). This improvement also led to a higher commit coverage for most of the projects, which we report in Table 3 and discuss in Section 6.3

Beside improving the identification of patches on mailing lists, we also had to individually check each mailing list for false positives, that is, whether the identified patches are actual patches that have been sent to the mailing list for to be reviewed. In some projects, when a commit is pushed to the main repository, also commits are sent to the developer mailing list as an automatically generated e-mail which contains the commit as a patch. Whereas most of the projects have separate mailing lists for such commit-bot e-mails, few projects (e.g. BUILDROOT and LLVM) use(d) the same mailing list for patch submission and automatic commit-bot e-mails. As `PaStA` would detect patches in such commit e-mails as patch revisions, this would distort our results. Hence, in such projects, we identified automatically generated commit e-mails on the mailing list (which is used for submitting patches) based on certain patterns[22] in the subject of such e-mails and removed all these e-mails from the extracted mailing-list data prior to running our analyses.

Acknowledgements

We thank Claus Hunsen for his support and fruitful discussions in early stages of this study. This work was supported by the German Research Foundation (AP 206/14-1). Ralf Ramsauer acknowledges support by the High-Tech Agenda of the Free State of

[21] Note that we also spotted lots of additional minor issues not mentioned here. Of course, we apply BASE64 decoding for BASE64 encoded e-mails, to name one example. Moreover, in some projects, we identified UUCP encoded e-mails (mostly created before 2000). However, as they only account for a very small fraction and since the extracted data for most of our subject projects starts after 2000 (see Table 2), we ignore UUCP encoded e-mails.

[22] Our code snippets for identifying commit-bot e-mails on the mailing lists of BUILDROOT and LLVM can be found on our supplementary website: https://se-sic.github.io/paper-patch-survival-analysis/

Bavaria. Furthermore, we thank the expert developers from our investigated projects for participating in our semi-structured interviews and for providing valuable insights into their integration processes.

Authorship Statement

1. **Christian Hechtl**: Conceptualization, Data Curation, Formal Analysis, Investigation, Methodology, Project Admin, Software, Validation, Visualization, Writing (original draft), Writing (review & editing)
2. **Thomas Bock**: Conceptualization, Data Curation, Formal Analysis, Investigation, Methodology, Project Admin, Software, Validation, Visualization, Writing (original draft), Writing (review & editing)
3. **Ralf Ramsauer**: Conceptualization, Data Curation, Formal Analysis, Funding Acquisition, Investigation, Methodology, Project Admin, Software, Validation, Visualization, Writing (original draft), Writing (review & editing)
4. **Wolfgang Mauerer**: Conceptualization, Formal Analysis, Funding Acquisition, Investigation, Methodology, Project Admin, Resources, Supervision, Visualization, Writing (original draft), Writing (review & editing)
5. **Sven Apel**: Conceptualization, Funding Acquisition, Investigation, Methodology, Project Admin, Supervision, Visualization, Writing (review & editing)

Data Availability

We provide a replication package including anonymized raw data, scripts, and tooling for our analyses on our supplementary website[23].

Competing Interests

The authors declare that they have no conflict of interest that is relevant to the content of this article.

References

Aman, H., Amasaki, S., Yokogawa, T., Kawahara, M.: A Survival Analysis of Source Files Modified by New Developers. In: Proc. Int. Conf. Product-Focused Software Process Improvement (PROFES), vol. 10611, pp. 80–88. Springer (2017)

Avelino, G., Constantinou, E., Valente, M.T., Serebrenik, A.: On the Abandonment and Survival of Open Source Projects: An Empirical Investigation. In: Proc. Int. Symposium Empirical Software Engineering and Measurement (ESEM), pp. 1–12. IEEE (2019)

Ait, A., Izquierdo, J.L.C., Cabot, J.: An Empirical Study on the Survival Rate of GitHub Projects. In: Proc. Int. Conf. Mining Software Repositories (MSR), pp. 365–375. ACM (2022)

[23] https://se-sic.github.io/paper-patch-survival-analysis/

Acuna, S.T., Juristo, N.: Assigning People to Roles in Software Projects. Software: Practice and Experience **34**(7), 675–696 (2004)

Asundi, J., Jayant, R.: Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study. In: Proc. Hawaii Int. Conf. System Sciences (HICSS), p. 166. IEEE (2007)

Alami, A., Pardo, R., Cohn, M.L., Wasowski, A.: Pull Request Governance in Open Source Communities. IEEE Transactions on Software Engineering (TSE) **48**(12), 4838–4856 (2022)

Agrawal, A., Rahman, A., Krishna, R., Sobran, A., Menzies, T.: We Don't Need Another Hero? The Impact of "Heroes" on Software Development. In: Proc. Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 245–253. ACM (2018)

Bock, T., Alznauer, N., Joblin, M., Apel, S.: Automatic Core-Developer Identification on GitHub: A Validation Study. ACM Transactions on Software Engineering and Methodology (TOSEM) **32**(6), 138 (2023)

Bosu, A., Carver, J.C.: Impact of Developer Reputation on Code Review Outcomes in OSS Projects: An Empirical Investigation. In: Proc. Int. Symposium Empirical Software Engineering and Measurement (ESEM), pp. 1–10. ACM (2014)

Bosu, A., Carver, J.C., Bird, C., Orbeck, J., Chockley, C.: Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft. IEEE Transactions on Software Engineering (TSE) **43**(1), 56–75 (2017)

Borgatti, S.P., Everett, M.G.: Models of core/periphery structures. Social Networks **21**(4), 375–395 (2000)

Brandes, U., Erlebach, T.: Network Analysis: Methodological Foundations. Springer (2005)

Bird, C., Gourley, A., Devanbu, P., Gertz, M., Swaminathan, A.: Mining Email Social Networks. In: Proc. Int. Conf. Mining Software Repositories (MSR), pp. 137–143. ACM (2006)

Bird, C., Gourley, A., Devanbu, P., Swaminathan, A., Hsu, G.: Open Borders? Immigration in Open Source Projects. In: Proc. Int. Conf. Mining Software Repositories (MSR). IEEE (2007)

Butler, S., Gamalielsson, J., Lundell, B., Brax, C., Sjöberg, J., Mattsson, A., Gustavsson, T., Feist, J., Lönroth, E.: On Company Contributions to Community Open Source Software Projects. IEEE Transactions on Software Engineering (TSE) **47**(7), 1381–1401 (2021)

Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., Hwang, D.-U.: Complex Networks: Structure and Dynamics. Physics reports **424**(4), 175–308 (2006)

Cairo, A.S., Figueiredo Carneiro, G., Monteiro, M.P.: The Impact of Code Smells on Software Bugs: A Systematic Literature Review. Information **9**(11), 273 (2018)

Cetin, H.A.: Identifying the Most Valuable Developers Using Artifact Traceability Graphs. In: Proc. Int. Symposium on Foundations of Software Engineering (FSE), pp. 1196–1198. ACM (2019)

Cheng, C., Li, B., Li, Z.-Y., Zhao, Y.-Q., Liao, F.-L.: Developer Role Evolution in Open Source Software Ecosystem: An Explanatory Study on GNOME. Journal of Computer Science and Technology (JCST) **32**(2), 396–414 (2017)

Cox, D.R., Oakes, D.: Analysis of Survival Data. Monographs on Statistics and Applied Probability, vol. 21. Chapman and Hall (1984)

Corbet, J.: How the Development Process Works. In: Linux Docs (2011). The Linux Foundation

Christian, J., Vu, A.N.: Task-based structures in open source software: revisiting the onion model. R&D Management **51**(1), 87–100 (2021)

Crowston, K., Wei, K., Li, Q., Howison, J.: Core and Periphery in Free/Libre and Open Source Software Team Communications. In: Proc. Hawaii Int. Conf. System Sciences (HICSS), pp. 118–118. IEEE (2006)

Chulani, S., Williams, C., Yaeli, A.: Software Development Governance and Its Concerns. In: Proc. Int. Workshop on Software Development Governance (SDG), pp. 3–6. ACM (2008)

De Laat, P.B.: Governance of Open Source Software: State of the Art. Journal of Management & Governance (JMG) **11**, 165–177 (2007)

Dey, T., Mockus, A.: Which Pull Requests Get Accepted and Why? A study of popular NPM Packages. CoRR **abs/2003.01153** (2020)

Diamantopoulos, T., Papamichail, M.D., Karanikiotis, T., Chatzidimitriou, K.C., Symeonidis, A.L.: Employing Contribution and Quality Metrics for Quantifying the Software Development Process. In: Proc. Int. Conf. Mining Software Repositories (MSR), pp. 558–562. ACM (2020)

Foundjem, A., Eghan, E.E., Adams, B.: Onboarding vs. Diversity, Productivity and Quality – Empirical Study of the OpenStack Ecosystem. In: Proc. Int. Conf. Software Engineering (ICSE). IEEE (2021)

Fowlkes, E.B., Mallows, C.L.: A Method for Comparing Two Hierarchical Clusterings. Journal of the American Statistical Association **78**(383), 553–569 (1983)

Fedorov, V., Mannino, F., Zhang, R.: Consequences of Dichotomization. Pharmaceutical Statistics **8**(1), 50–61 (2009)

Foucault, M., Palyart, M., Blanc, X., Murphy, G.C., Falleri, J.-R.: Impact of developer turnover on quality in open-source software. In: Proc. Int. Symposium on Foundations of Software Engineering (FSE), pp. 829–841. ACM (2015)

Guzzi, A., Bacchelli, A., Lanza, M., Pinzger, M., Van Deursen, A.: Communication in Open Source Software Development Mailing Lists. In: Proc. Int. Conf. Mining Software Repositories (MSR), pp. 277–286. IEEE (2013)

Gall, H., Hajek, K., Jazayeri, M.: Detection of Logical Coupling based on Product Release History. In: Proc. Int. Conf. Software Maintenance (ICSM), pp. 190–198. IEEE (1998)

Gharehyazie, M., Posnett, D., Vasilescu, B., Filkov, V.: Developer initiation and social interactions in OSS: A case study of the Apache Software Foundation. Empirical Software Engineering (EMSE) **20**, 1318–1353 (2015)

Hechtl, C., Joblin, M., Apel, S.: Is Perceived Gender Related to Contributions and Standing in Open-Source Software Projects? Empirical Software Engineering (EMSE) **30**(5), 123 (2025)

Haider, S., Khalil, W., Al-Shamayleh, A.S., Akhunzada, A., Gani, A.: Risk Factors and Practices for the Development of Open Source Software From Developers' Perspective. IEEE Access **11**, 63333–63350 (2023)

He, P., Li, B., Huang, Y.: Applying Centrality Measures to the Behavior Analysis of Developers in Open Source Software Community. In: Proc. Int. Conf. Cloud and Green Computing (CGC), pp. 418–423. IEEE (2012)

Izquierdo, J.L.C., Cabot, J.: Enabling the Definition and Enforcement of Governance Rules in Open Source Systems. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 505–514. IEEE (2015)

Jiang, Y., Adams, B., German, D.M.: Will My Patch Make It? And How Fast? Case Study on the Linux Kernel. In: Proc. Int. Conf. Mining Software Repositories (MSR), pp. 101–110. IEEE (2013)

Joblin, M., Apel, S., Hunsen, C., Mauerer, W.: Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 164–174. IEEE (2017)

Jiang, Y., Adams, B., Khomh, F., German, D.M.: Tracing Back the History of Commits in Low-Tech Reviewing Environments: A Case Study of the Linux Kernel. In: Proc. Int. Symposium Empirical Software Engineering and Measurement (ESEM), pp. 1–10. ACM (2014)

Joblin, M., Apel, S., Mauerer, W.: Evolutionary Trends of Developer Coordination: A Network Approach. Empirical Software Engineering **22**(4), 2050–2094 (2017)

Joblin, M., Eckl-Ganser, B., Bock, T., Schmid, A., Siegmund, J., Apel, S.: Hierarchical and Hybrid Organizational Structures in Open-Source Software Projects: A Longitudinal Study. ACM Transactions on Software Engineering and Methodology (TOSEM) **32**(4), 86 (2023)

Joblin, M., Mauerer, W., Apel, S., Siegmund, J., Riehle, D.: From Developer Networks to Verified Communities: A Fine-Grained Approach. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 563–573. IEEE (2015)

Khatoonabadi, S., Abdellatif, A., Costa, D.E., Shihab, E.: Predicting the First Response Latency of Maintainers and Contributors in Pull Requests. CoRR **abs/2311.07786** (2023)

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W.: Investigating Code Review Quality: Do People and Participation Matter? In: Proc. Int. Conf. Software Maintenance and Evolution (ICSME), pp. 111–120. IEEE (2015)

Kononenko, O., Baysal, O., Godfrey, M.W.: Code Review Quality: How Developers See It. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 1028–1038. ACM (2016)

Käfer, V., Graziotin, D., Bogicevic, I., Wagner, S., Ramadani, J.: Communication in Open-Source Projects – End of the E-mail Era? In: Companion Volume ICSE, pp. 242–243. ACM (2018)

Kroah-Hartman, G.: Why kernel development still uses email. Linux Weekly News (LWN) (2016)

Kleinbaum, D.G., Klein, M.: Survival Analysis: A Self-Learning Text, 3rd edn. Springer (2012)

Koru, A.G., Zhang, D., Liu, H.: Modeling the Effect of Size on Defect Proneness for Open-Source Software. In: Proc. Int. Conf. Predictive Models in Software Engineering (PROMISE). IEEE (2007)

Lin, B., Robles, G., Serebrenik, A.: Developer Turnover in Global, Industrial Open Source Projects: Insights from Applying Survival Analysis. In: Proc. Int. Conf. Global Software Engineering (ICGSE), pp. 66–75. IEEE (2017)

Lakhani, K.R., Wolf, R.G.: Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. Perspectives on Free and Open Source Software **1**, 3–22 (2005)

Liu, B., Zhang, H., Ma, W., Kuang, H., Yang, Y., Xu, J., Gao, S., Gao, J.: Mining Pull Requests to Detect Process Anomalies in Open Source Software Development.

In: Proc. Int. Conf. Software Engineering (ICSE), pp. 194–119413. ACM (2024)

Mannan, U.A., Ahmed, I., Jensen, C., Sarma, A.: On the Relationship between Design Discussions and Design Quality: A Case Study of Apache Projects. In: Proc. Europ. Software Engineering Conf. and the Int. Symposium Foundations of Software Engineering (ESEC/FSE), pp. 543–555. ACM (2020)

Markus, M.L.: The Governance of Free/Open Source Software Projects: Monolithic, Multidimensional, or Configurational? Journal of Management & Governance (JMG) **11**, 151–163 (2007)

Maruping, L.M., Daniel, S.L., Cataldo, M.: Developer Centrality and the Impact of Value Congruence and Incongruence on Commitment and Code Contribution Activity in Open Source Software Communities. MIS Quarterly **43**(3), 951–976 (2019)

MacKenzie, D., Eggert, P., Stallman, R.: Comparing and Merging Files with GNU Diff and Patch, 2nd edn. Network Theory Limited (2003)

Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology (TOSEM) **11**(3), 309–346 (2002)

Michlmayr, M., Hunt, F., Probert, D.: Quality Practices and Problems in Free Software Projects. In: Int. Conf. Open Source Systems (OSS), pp. 24–28. IEEE (2005)

Mauerer, W., Jaeger, M.C.: Open Source Engineering Processes. it-Information Technology **55**(5), 196–203 (2013)

Mauerer, W., Joblin, M., Tamburri, D.A., Paradis, C., Kazman, R., Apel, S.: In Search of Socio-Technical Congruence: A Large-Scale Longitudinal Study. IEEE Transactions on Software Engineering (TSE) **48**(8), 3159–3184 (2022)

McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the QT, VTK, and ITK Projects. In: Proc. Int. Conf. Mining Software Repositories (MSR), pp. 192–201. ACM (2014)

McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. Empirical Software Engineering **21**(5), 2146–2189 (2016)

Meneely, A., Williams, L.: Socio-technical Developer Networks: Should We Trust Our Measurements? In: Proc. Int. Conf. Software Engineering (ICSE), pp. 281–290. ACM (2011)

Nurolahzade, M., Nasehi, S.M., Khandkar, S.H., Rawal, S.: The Role of Patch Review

in Software Evolution: An Analysis of the Mozilla Firefox. In: Proc. Int. Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution (IWPSE-EVOL), pp. 9–18. ACM (2009)

Ortega, F., Izquierdo-Cortazar, D.: Survival Analysis in Open Development Projects. In: ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS), pp. 7–12. IEEE (2009)

Oliva, G.A., Santana, F.W., Oliveira, K.C.M., Souza, C.R.B., Gerosa, M.A.: Characterizing Key Developers: A Case Study With Apache Ant. In: Proc. Int. Conf. Collaboration and Technology (CRIWG), pp. 97–112. Springer (2012)

O'Mahony, S.: The Governance of Open Source Initiatives: What Does It Mean to Be Community Managed? Journal of Management & Governance (JMG) **11**, 139–150 (2007)

Pinto, G., Steinmacher, I., Gerosa, M.A.: More Common than You Think: An In-Depth Study of Casual Contributors. In: Int. Conf. Software Analysis, Evolution, and Reengineering (SANER), pp. 112–123. IEEE (2016)

Qiu, H.S., Nolte, A., Brown, A., Serebrenik, A., Vasilescu, B.: Going Farther Together: The Impact of Social Capital on Sustained Participation in Open Source. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 688–699. IEEE (2019)

Ramsauer, R., Bulwahn, L., Lohmann, D., Mauerer, W.: The Sound of Silence: Mining Security Vulnerabilities from Secret Integration Channels in Open-Source Projects. In: Proc. Conf. Cloud Computing Security Workshop (CCSW), pp. 147–157. ACM (2020)

Rigby, P.C., German, D.M., Cowen, L., Storey, M.-A.: Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. ACM Transactions on Software Engineering and Methodology (TOSEM) **23**(4), 35–13533 (2014)

Rigby, P.C., Germán, D.M., Storey, M.D.: Open Source Software Peer Review Practices: A Case Study of the Apache Server. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 541–550. ACM (2008)

Ramsauer, R., Lohmann, D., Mauerer, W.: The List is the Process: Reliable Pre-Integration Tracking of Commits on Mailing Lists. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 807–818. IEEE (2019)

Rich, J.T., Neely, J.G., Paniello, R.C., Voelker, C.C.J., Nussenbaum, B., Wang, E.W.: A Practical Guide to Understanding Kaplan-Meier Curves. Otolaryngology–Head and Neck Surgery **143**(3), 331–336 (2010)

Rigby, P.C., Storey, M.-A.: Understanding Broadcast Based Peer Review on Open Source Software Projects. In: Proc. Int. Conf. Software Engineering (ICSE), pp.

541–550. ACM (2011)

Schilling, A.: What Do We Know about FLOSS Developers' Attraction, Retention, and Commitment? A Literature Review. In: Proc. Hawaii Int. Conf. System Sciences (HICSS), pp. 4003–4012. IEEE (2014)

Shaikh, M., Henfridsson, O.: Governing open source software through coordination processes. Information and Organization **27**(2), 116–135 (2017)

Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. IEEE Transactions on Software Engineering (TSE) **37**(6), 772–787 (2011)

Storey, M.-A., Singer, L., Figueira Filho, F., Zagalsky, A., German, D.M.: How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. IEEE Transactions on Software Engineering (TSE) **43**(2), 185–204 (2017)

Shibuya, B., Tamai, T.: Understanding the Process of Participating in Open Source Communities. In: ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS), pp. 1–6. IEEE (2009)

Tao, Y., Han, D., Kim, S.: Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In: Proc. Int. Conf. Software Maintenance and Evolution (ICSME), pp. 271–280 (2014). IEEE

Tamburri, D.A., Lago, P., Vliet, H.v.: Organizational Social Structures for Software Engineering. ACM Computing Surveys **46**(1), 1–35 (2013)

Terceiro, A., Rios, L.R., Chavez, C.: An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects. In: Proc. Brazilian Symposium on Software Engineering (SBES), pp. 21–29. IEEE (2010)

Torres, M.R.M., Toral, S.L., Perales, M., Barrero, F.: Analysis of the Core Team Role in Open Source Communities. In: Proc. Int. Conf. Complex, Intelligent, and Software Intensive Systems (CISIS), pp. 109–114. IEEE (2011)

Tan, X., Zhou, M.: How to Communicate when Submitting Patches: An Empirical Study of the Linux Kernel. Proceedings of the ACM on Human-Computer Interaction (HCI) **3**(CSCW), 108–110826 (2019)

Tan, X., Zhou, M., Fitzgerald, B.: Scaling Open Source Communities: An Empirical Study of the Linux Kernel. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 1222–1234. ACM (2020)

Wirth, N.: Program Development by Stepwise Refinement. Communications of the ACM **14**(4), 221–227 (1971)

Weißgerber, P., Neu, D., Diehl, S.: Small Patches Get In! In: Proc. Int. Conf. Mining Software Repositories (MSR), pp. 67–76. ACM (2008)

Wang, J., Shih, P.C., Wu, Y., Carroll, J.M.: Comparative Case Studies of Open Source Software Peer Review Practices. Information and Software Technology (IST) **67**, 1–12 (2015)

Wiese, I.S., Silva, J., Steinmacher, I., Treude, C., Gerosa, M.A.: Who is Who in the Mailing List? Comparing Six Disambiguation Heuristics to Identify Multiple Addresses of a Participant. In: Proc. Int. Conf. Software Maintenance and Evolution (ICSME), pp. 345–355. IEEE (2016)

Yamashita, K., McIntosh, S., Kamei, Y., Hassan, A.E., Ubayashi, N.: Revisiting the Applicability of the Pareto Principle to Core Development Teams in Open Source Software Projects. In: Proc. Int. Workshop on Principles of Software Evolution (IWPSE), pp. 46–55. ACM (2015)

Yin, L., Zhang, X., Filkov, V.: On the Self-Governance and Episodic Changes in Apache Incubator Projects: An Empirical Study. In: Proc. Int. Conf. Software Engineering (ICSE), pp. 678–689. IEEE (2023)

Zhang, X., Yu, Y., Gousios, G., Rastogi, A.: Pull Request Decisions Explained: An Empirical Overview. IEEE Transactions on Software Engineering (TSE) **49**(2), 849–871 (2023)

Zhu, J., Zhou, M., Mockus, A.: Effectiveness of Code Contribution: From Patch-based to Pull-request-based Tools. In: Proc. Int. Symposium on Foundations of Software Engineering (FSE), pp. 871–882. ACM (2016)

Zimmermann, T., Zeller, A., Weißgerber, P., Diehl, S.: Mining Version Histories to Guide Software Changes. IEEE Transactions on Software Engineering (TSE) **31**(6), 429–445 (2005)