Emerging Organizational Patterns in Evolving Open-Source Software Projects

An Analysis of Developer Activity and Coordination

Thomas Bock

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

Saarbrücken, 2024

UNIVERSITÄT
DES
SAARLANDES

Abstract

Many popular and widely-used software projects are organized and developed as open-source software (OSS) projects. Their source code is publicly available and everyone can contribute. Thus, many OSS projects are able to attract a high number of contributors, who develop the source code of the project or participate in the public communication channels of the project. It is almost folklore that, when multiple developers contribute to the source code of a project simultaneously, proper coordination between the developers is necessary to avoid unexpected interactions between the simultaneously changed source-code parts and to reduce the risk of introducing new bugs. As indicated by previous research, software bugs are often caused by a lack of coordination and by problems in the organizational structure of the project. However, while there is a clearly defined organizational hierarchy among the developers in commercial, closed-source software projects, this does not necessarily apply to OSS projects, which sometimes are seen as self-organizing communities. Consequently, to improve developer coordination in OSS projects, it is essential to understand the organizational structure of these projects.

For this purpose, we analyze the organizational structure of OSS projects and how it evolves over time, by means of five different empirical studies on widely-used and well-known OSS projects. This way, we aim at obtaining an understanding of developer activity and coordination, which shall serve as a first step toward improved coordination processes and, in turn, toward less flawed software products. In particular, we address three different aspects: the evolution of developer collaboration and communication, the identification of developer roles, and the relation between organizational events and developer-network characteristics.

First, to obtain insights into the evolution of *developer collaboration* (i.e., mutual editing of a source-code artifact) and *developer communication* (i.e., mutual commenting on the same issue or in the same mailing-list thread), we investigate how collaboration and communication are temporally and content-wise related, as previous work has proposed that collaboration and communication activities should align due to their common tasks. Therefore, we take a conversation-based view and develop a method to measure the synchronicity of collaboration and communication activities. Moreover, we also take a developer-network perspective to examine whether developers form stable group structures and how they evolve over time. To that aim, we devise a modular, multi-step measuring and modeling framework based on canonical tensor decomposition to detect developer groups and predict interactions among developers. This way, we are able to identify stable groups as well as distortions in the group structure. In addition, we propose a method to detect hierarchical structures, and we report on insights from an empirical study regarding the presence and evolution of hierarchical structures in developer networks as well as regarding the roles developers take with respect to their position in the hierarchy. These insights provide the basis for future investigations on the organizational structure of OSS projects and shall help to guide project governance.

Second, as developers in OSS projects perform different tasks and take different roles, *core developers* play an important part with respect to coordination within the organizational structure of an OSS project. While *peripheral developers* contribute only occasionally to the project (e.g., provide a bug fix), core developers steadily contribute and work on central maintenance tasks. While there is a growing corpus of research that attempts to identify core developers, the performance of such identification methods is largely unclear. Therefore, we propose an automatic method to identify core developers based on privileged role permissions in GitHub issues and pull requests. In a validation study, we validate our derived set of privileged developers with official, publicly available maintainers lists. Moreover, we perform an assessment of the accuracy of state-of-the-art developer-role classification methods. Beside devising an automated method for core developer identification, our results help researchers and practitioners to choose an appropriate classification method.

Third, we investigate the relation between *organizational events* (such as the creation of forks or company acquisitions) and *developer-network characteristics*, as organizational events can dramatically influence the organizational structure and, thus, the collaboration and communication between developers of OSS projects. An analysis of developer-network characteristics can reveal potentially unobserved changes in the organizational structure. Based on observations from an exploratory study on eight case studies, we develop hypotheses regarding which network characteristics are promising indicators for detecting organizational events that affect the structure of developer networks. This way, we lay the foundations for future work on developing measures to detect organizational events that may potentially threaten the sustainability of an OSS project.

In summary, our work comprehensively studies developers' programming and communication activities in OSS projects. For this purpose, we provide multiple methods to analyze the collaboration and communication activities of developers with respect to evolutionary organizational structures. We demonstrate the applicability of the proposed methods and provide valuable insights from different empirical studies on the coordination processes in OSS projects. The insights we obtain shall enable devising guidelines and developing improved processes for developer coordination in the future.

Zusammenfassung

Viele populäre und weit verbreitete Softwareprojekte werden als Open-Source-Software-Projekte (OSS-Projekte) organisiert und entwickelt, deren Quellcode öffentlich zugänglich ist und zu dem jeder beitragen kann. Daher weisen viele OSS-Projekte eine hohe Anzahl an Mitwirkenden auf, die den Quellcode des Projekts entwickeln oder sich an den öffentlichen Kommunikationskanälen des Projekts beteiligen. Wenn mehrere Entwickler gleichzeitig zum Quellcode eines Projekts beitragen, ist eine angemessene Koordinierung zwischen den Entwicklern notwendig, um unerwartete Interaktionen zwischen den gleichzeitig geänderten Teilen des Quellcodes zu vermeiden und das Risiko der Einführung neuer Fehler zu verringern. Wie frühere Untersuchungen gezeigt haben, werden Softwarefehler häufig durch mangelnde Koordinierung zwischen den Entwicklern und durch Probleme in der Organisationsstruktur des Projekts verursacht. Während es in kommerziellen, nicht-quelloffenen Softwareprojekten eine klar definierte organisatorische Hierarchie unter den Entwicklern gibt, gilt dies nicht unbedingt für OSS-Projekte, die manchmal als selbstorganisierende Gemeinschaften angesehen werden. Um die Koordinierung der Entwickler in OSS-Projekten zu verbessern, ist es daher wichtig, die Organisationsstruktur dieser Projekte zu verstehen.

Zu diesem Zweck analysieren wir in fünf verschiedenen empirischen Studien an weit verbreiteten und bekannten OSS-Projekten deren Organisationsstruktur und wie sie sich im Laufe der Zeit entwickelt hat. Auf diese Weise wollen wir ein Verständnis über die Entwickleraktivität und -koordinierung erlangen, das ein erster Schritt zu verbesserten Koordinierungsprozessen sein soll und damit zu weniger fehlerhaften Softwareprodukten führen soll. Wir befassen uns insbesondere mit drei verschiedenen Aspekten: der Entwicklung der Zusammenarbeit und Kommunikation von Entwicklern über die Zeit, der Identifizierung von Entwicklerrollen und der Beziehung zwischen organisatorischen Ereignissen und Eigenschaften von Entwicklernetzwerken.

1. Um Einblicke in die Entwicklung der *Zusammenarbeit* (d.h., die wechselseitige Bearbeitung eines Quellcode-Artefakts) und der *Kommunikation* (d.h., das wechselseitige Kommentieren im selben Issue oder im selben Mailinglisten-Thread) von Software-Entwicklern zu erhalten, untersuchen wir zunächst, wie Zusammenarbeit und Kommunikation zeitlich und inhaltlich miteinander verbunden sind. Frühere Arbeiten haben vorgeschlagen, dass die Zusammenarbeit und Kommunikation von Entwicklern aufgrund ihrer gemeinsamen Aufgaben aufeinander abgestimmt werden sollten. Daher nehmen wir eine konversationsbasierte Sichtweise ein und entwickeln eine Methode, um die Synchronität von Zusammenarbeit und Kommunikation zu messen. Darüber hinaus untersuchen wir mit Hilfe von Entwicklernetzwerken, ob Entwickler stabile Gruppenstrukturen bilden und wie diese sich im Laufe der Zeit entwickeln. Zu diesem Zweck erarbeiten wir ein modulares, mehrstufiges Mess- und Modellierungs-Framework basierend auf kanonischer Tensor-Dekomposition, um Entwickler-

gruppen zu erkennen und Interaktionen zwischen den Entwicklern vorherzusagen. Auf diese Weise sind wir in der Lage, sowohl stabile Gruppen als auch Verzerrungen in der Gruppenstruktur zu identifizieren. Darüber hinaus schlagen wir eine Methode vor, um hierarchische Strukturen zu erkennen. Damit gewinnen wir aus einer empirischen Studie Erkenntnisse über das Vorhandensein und die Entwicklung von hierarchischen Strukturen in Entwicklernetzwerken sowie über die Rollen, die Entwickler in Bezug auf ihre Position in der Hierarchie einnehmen. Diese Erkenntnisse bieten die Grundlage für zukünftige Untersuchungen zur Organisationsstruktur von OSS-Projekten und sollen dabei helfen, die Projektführung zu steuern.

2. Da die Entwickler in OSS-Projekten unterschiedliche Aufgaben und Rollen übernehmen, spielen die *Kernentwickler* eine wichtige Rolle bei der Koordinierung innerhalb der Organisationsstruktur eines OSS-Projekts. Während *periphere Entwickler* nur gelegentlich zum Projekt beitragen (z.B. einen Bugfix liefern), tragen Kernentwickler kontinuierlich zum Projekt bei und übernehmen zentrale Wartungsaufgaben. Es gibt zwar eine wachsende Zahl von Forschungsarbeiten, die versuchen, die Kerntwickler zu identifizieren, aber die Genauigkeit solcher Identifizierungsmethoden ist weitgehend unklar. Aus diesem Grund entwickeln wir eine automatische Methode zur Identifizierung von Kernentwicklern, die auf privilegierten Rollen-Berechtigungen in GitHub-Issues und Pull-Requests basiert. In einer Validierungsstudie gleichen wir unsere extrahierte Gruppe von privilegierten Entwicklern mit offiziellen, öffentlich zugänglichen Maintainer-Listen ab. Außerdem führen wir eine Bewertung der Genauigkeit etablierter Methoden zur Klassifizierung von Entwicklerrollen durch. Neben der Entwicklung einer automatisierten Methode zur Identifizierung von Kernentwicklern sollen unsere Ergebnisse Forschern und Praktikern bei der Auswahl einer geeigneten Klassifizierungsmethode helfen.

3. Schließlich untersuchen wir die Beziehung zwischen *organisatorischen Ereignissen* (wie z.B. der Gründung von Abspaltungen eines Projekts oder Firmenübernahmen) und den *Eigenschaften von Entwicklernetzwerken*, da organisatorische Ereignisse die Organisationsstruktur von OSS-Projekten und somit auch die Zusammenarbeit der Entwickler und die Kommunikation zwischen den Entwicklern dramatisch beeinflussen können. Eine Analyse des Entwicklernetzwerks eines Projekts ist dazu in der Lage, Veränderungen in der Organisationsstruktur aufzudecken, die bislang potenziell unbeobachtet waren. Auf der Grundlage von Beobachtungen aus einer explorativen Studie mit acht Fallstudien entwickeln wir Hypothesen darüber, welche Netzwerkeigenschaften vielversprechende Indikatoren sein könnten für die Erkennung organisatorischer Ereignisse, die die Struktur von Entwicklernetzwerken beeinflussen. Auf diese Weise legen wir den Grundstein für die zukünftige Entwicklung von Maßnahmen zur Erkennung jener organisatorischer Ereignisse, welche möglicherweise die nachhaltige Entwicklung eines OSS-Projekts gefährden können.

Diese Arbeit untersucht die Programmier- und Kommunikationsaktivitäten von Entwicklern in OSS-Projekten auf verschiedene Weise. Einerseits erarbeiten wir mehrere Methoden zur Analyse der Zusammenarbeit und Koordinierung von Entwicklern im Hinblick auf evolutionäre Organisationsstrukturen, und andererseits zeigen wir auch die Anwendbarkeit der erarbeiteten Methoden mit Hilfe verschiedener empirischer Studien. Die Erkenntnisse, die wir aus unseren Studien gewinnen, sollen es schließlich ermöglichen, Richtlinien zu erarbeiten und verbesserte Prozesse für die zukünftige Entwicklerkoordinierung zu entwickeln.

Acknowledgments

Writing a doctoral thesis is a difficult endeavor, lasting for several years, and, thus, requires the support of many people. I am grateful for every person I got to know during this endeavor, as well as for having friends and colleagues who helped me in all kinds of situations.

First and foremost, I would like to thank my parents, who have always had my back all the time and who have encouraged me on my way on pursuing a doctoral degree.

A heartfelt thank you goes to my supervisor Sven Apel, who gave me the opportunity to conduct interesting and exciting academic research at his Chair of Software Engineering within a group of enthusiastic researchers and to write this doctoral thesis in a cordial and open environment. Sven also guided me well through the various challenges that arise when doing research and also by sharing his wealth of experience and ideas.

I also would like to thank Janet Siegmund, whom I got to know when I was an undergraduate student many years ago and who had initially introduced me to the research group of Sven Apel. It was a pleasure to also collaborate with her and have her as a co-author of research papers. In addition, there are also several other co-authors of my research papers whom I would like to thank: Many thanks go to Mitchell Joblin, who supported me over many years with his fruitful comments, his methodological experience, and his enthusiasm for research. He always provided helpful feedback and enriched my work with inspiring ideas. Moreover, I would also like to thank Angelika Schmid for sharing her expertise in statistics and also for sharing any number of excellent ideas. Furthermore, I would like to thank Barbara Eckl-Ganser and Claus Hunsen, with whom I had the pleasure of working together in the early years of my research endeavor and with whom we started to build up the data-extraction and data-preprocessing toolchain that serves as the technical basis for this thesis.

Many thanks go to my friends and colleagues at the Chair of Software Engineering. Special thanks go to Christian Kaltenecker, who accompanied me from the beginning of my undergraduate studies and always supported me in whatever respect through this long period of time until today. I would also like to thank Florian Sattler for his assistance with regard to technical matters as well as for many fruitful discussions and convivial evenings. Moreover, I would like to thank my office colleague Christian Hechtl, who was always available for spontaneous discussions. In addition, I would like to thank Christian Kaltenecker, Florian Sattler, and Christian Hechtl for proofreading parts of this thesis and providing valuable feedback. Besides, I would like to thank Kallistos Weis and Sebastian Böhm for providing valuable feedback on selected visualizations and figures. I am also grateful for the many other colleagues who have accompanied me on the way to this doctoral thesis. Further thanks go to Nils Alznauer for his support in one of the empirical studies. Also the many student research assistants who have contributed to the toolchain deserve my thanks, especially Niklas Schneider, who helped me over several years quickly and reliably at any time.

Additional thanks go to my second reviewer Bogdan Vasilescu for agreeing to read and review this thesis, which is a time-consuming task, and for making the effort to attend my colloquium on site.

Finally, I also would like to thank Eva Reichhart and Friederike Repplinger for their administrative support as well as Armin Größlinger for providing and suggesting solutions to technical challenges of any kind when analyzing huge amounts of data. All three always had a sympathetic ear for me and helped me with any regards.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	application programming interface
AUC	area under curve
CLA	contributor licensing agreement
CSV	comma-separated-value
DTW	dynamic time warping
ER	Erdős-Rényi
FDR	false-discovery rate
FP	false positives

FPR	false-positive rate
FN	false negatives
LOC	lines of code
OSS	open-source software
PR	pull request
RAM	random-access memory
REST	representational state transfer
RMSE	root-mean-square error
RO	research objective
ROC	receiver operating characteristic
RQ	research question
TP	true positives
TPR	true-positive rate
TN	true negatives
VCS	version-control system
UTC	coordinated universal time

Introduction

1

Software engineering is an increasingly important discipline in the era of digitalization. Many people all over the world rely on software in their daily life, being it in their spare time (e.g., entertainment applications), for their jobs (e.g., web services), or for organizational tasks (e.g., services of government offices). In the near future, software will become even more essential in everyday life in the face of upcoming developments such as artificial intelligence, robotics, or autonomous cars. That is, people depend on software [Far19] (e.g., life without online payment or online stores is not imaginable any more), and people are influenced by software (e.g., through personalized advertising [CVL+22; RU20; WPA+18] or automatically proposed news articles[1] [ZW17]). For this reason, it is important for software developers to develop and deploy high-quality and highly reliable software to fulfill people's requirements and also to prevent software causing serious damage. This becomes even more important when software interacts with humans, such as self-driving cars, which need to watch out for pedestrians, or surgery robots, which need to perform medical surgeries without causing damage. Only a small deviation from the expected behavior that is caused by a bug in the software can already be very harmful. In the past, several accidents happened in which people died because of software bugs: A pedestrian was killed by a self-driving car, several people were killed or injured by software-caused overdoses of radiation, and hundreds of airplane passengers died because of software-caused crashes of airplanes, etc.[2] Previous research has shown that software bugs, in general, are often caused (among others) by chaotic project management, missing quality control, lack of coordination, or problems in the organizational structure of the project [HKA+23; LMV+14; MHP05; ZCC+19].

As software is often created by several parties, proper coordination among software developers is necessary, to avoid as many bugs as possible [BCD+12; MHP05; ZCC+19], and to improve software quality [MHP05; MKA+16; SFS+17]. For example, in the Linux kernel development project (a large open-source operating system), more than 10 000 developers contributed to the source code already. Hence, it is not uncommon that multiple developers independently contributed to interrelated source-code fragments. If not coordinated, unintentional side effects, bugs, and vulnerabilities can be introduced into the code base [CH13; GHP99; KSD11; SMW+11]. Therefore, it is imperative that developers *coordinate* before they change related source-code fragments [CWH+06; HMR06; MC90]. Depending on the organizational structure of a software project and the used means of communication, coordination

1 Paul Farhi: "A news site used AI to write articles. It was a journalistic disaster.", 2023: https://www.washingtonpost.com/media/2023/01/17/cnet-ai-articles-journalism-corrections/ (accessed at 2023-06-29)

2 Keri Savoca: "When Software Kills", 2019: https://medium.com/swlh/when-software-kills-ab6f48a15825/ (accessed at 2023-05-05)

requirements among developers are not always fulfilled. Fulfilling the coordination requirements is expensive and could also end up in an increased maintenance overhead [WXY+18]. Consequently, tools are necessary that assist developers to improve the project workflow and to reduce maintenance costs. In particular, tools must guide developers for improving collaboration and coordination within software projects.

Nowadays, many of the everyday-life software products are developed by open-source software (OSS) projects (e.g., web browsers such as Firefox, office software such as LibreOffice, or pandemic contact-tracing mobile apps such as the Corona Warn App). Even beyond that, OSS projects are widely used for software development [RSA+19; Rie19]. A benefit of OSS projects is that their source-code is publicly available, reusable, and testable, such that voluntary developers can contribute to the project (e.g., search or fix a bug, implement enhancements, etc.) [BR03; KU12; MČ15; MD14]. While there is typically a clearly defined organizational hierarchy among software developers in commercial, closed-sourced projects, this does not necessarily apply to OSS projects [HC14; LBG+16; NKU17; PAR14; SSR02]. Because of the public availability of source code and corresponding communication channels, and because of the participation of volunteers, OSS projects sometimes are even seen as self-organizing communities [CLW+07; MFT02; Yu08]. This is also reflected in the different kinds of developers that contribute to OSS projects: On the one hand, there are developers who take central roles in the project, often called "core" developers, lead developers, or maintainers. On the other hand, there are developers who contribute only occasionally to the project, called "peripheral" developers [JAH+17; MFH02; NYN+02]. As peripheral developers usually do not contribute regularly and are typically only active for a short period of time, they have to be up to date regarding the source-code changes of other developers and, thus, should coordinate with the core developers to avoid unwanted changes. As the number of participating developers often increases over time, it is even more difficult to maintain large software projects and check whether all changes of all developers still guarantee the intended behavior of the software [GHI21; SMS16]. Due to the global digitalization, the number of software developers increases steadily, which also affects global interactions of developers. Also, different cultural backgrounds and custom habits of developers may play a vital role when developers communicate with each other [LK06; MLB18; ZLS22]. As a consequence, it is essential to understand the organizational structure of software projects to improve developer coordination and ease software maintenance in large OSS projects.

In this thesis, we shed light on developers' programming and communication activities and their coordination, how these activities are related to each other, and how they change over time. By means of multiple empirical studies on several widely-used and well-known OSS projects, we analyze the organizational structure of OSS projects and their evolution with regard to various different factors, such as developer roles or external events that influence the organizational structure. Through our studies, we provide valuable insights into the development and organizational processes of various OSS projects, which shall serve as a basis for devising guidelines and improved processes for developer coordination in the future.

1.1 Goals of the Thesis

In this thesis, we pursue the overall goal of obtaining an understanding of the organizational structures in OSS projects. Only if these structures are understood, informed recommendations on how to improve coordination processes and organizational structures in OSS projects can be made, which, in turn, is supposed to lead to an improvement of software quality [CH05; CH13; NMB08; SFD05]. To this end, we investigate the *collaboration processes* of developers with respect to their source-code changes, as well as the *communication processes* of developers on the major communication channels of a project, such as mailing lists or issue trackers. More generally speaking, we analyze social and socio-technical aspects of software engineering, as software development is not only a technical activity but also involves human interaction in a technical setting. In particular, we analyze the coding activity as well as the communication activity in OSS projects, using various data analysis techniques and methods from social-network analysis (since interacting software developers also form a social network and a network perspective can reveal information that is not directly observable from the plain data). This way, we aim at identifying and extracting organizational patterns that can provide new insights into the underlying development processes.

We subdivide our overall research objective (RO) into three sub-objectives:

RO$_1$: *Understand the evolution of collaboration and communication processes, how they are temporally and content-wise related, and how the underlying group structures evolve.*
Over time, the organizational structure in an OSS project may change, since also the number of developers changes and developers take on different tasks during the course of their participation in a project. At project initiation, when only a few developers are contributing, the structures may be different than for a project that lasts already for several years. Different group structures may arise or vanish over time, for instance, when new core developers join or new development tasks emerge, or when established core developers leave the project. In such cases, communicating project knowledge from established to new developers is imperative, to avoid a loss of implicit project knowledge during the changes of the group structure. But not only global project evolution matters, also the temporal and content-wise alignment of collaboration and communication activity is important. According to the "mirroring hypothesis" [CB16; KCD12], organizational structures in collaboration and coordination should align due to their common tasks. Previous work assumed that the collaboration and communication activities of two developers are aligned on common tasks when they collaborate on the source code and communicate at the same time [XF14]. However, using such an assumption, we do not know whether the temporally aligned communication of the developers covers the topics of their joint coding activities. This is why also a content-wise relation between collaboration on the source code and communication should be present to fulfill the "mirroring hypothesis". Therefore, we aim at obtaining an understanding of how collaboration and communication in OSS projects are temporally and content-wise related and how the corresponding organizational structures evolve over time.

RO$_2$: *Devise and investigate means for identifying core developers and potential candidates for maintainers.*
As developers in OSS projects take different roles, identifying these roles is pivotal for understanding organizational structures, especially with respect to leadership roles [CLL+17; Lon06; YHH04]. However, in OSS projects, core developers are not always clearly distinguishable from peripheral developers, which is why classification methods have been devised in the literature to identify core developers. Nevertheless, different state-of-the-art developer-classification methods may lead to different results. Hence, investigating the state-of-the-art classification methods and assessing their accuracy leads to identifying their strengths and limitations. Knowing about the strengths and limitations of developer-classification methods helps understanding the developers' roles and can also be used for selecting an appropriate classification method to identify potential candidates for core developers in a project. Identifying potential candidates for core developers, in turn, may help stabilize the project's organizational structure, avoiding organizational vacuum and knowledge loss. For this purpose, we aim at devising a method to identify core developers and at informing researchers and practitioners about the performance of state-of-the-art methods for identifying core developers.

RO$_3$: *Investigate how organizational events influence the organizational structure of an OSS project.*
Organizational events, such as emerging forks, core developers who leave, or a company taking over a project, can influence the project's organizational structure dramatically. For instance, when a core developer leaves a project, this could imply a loss of project knowledge if the remaining developers do not know about particular specifics of the project that only the leaving developer knows. This, in turn, could lead to the introduction of bugs in the future. Investigating the organizational structure and developer-network characteristics before and after such events could reveal the existence of such specifics, which could be used to start counteractions or to develop an early-warning mechanism to avoid such problems in other projects or future situations. We aim at obtaining an understanding of the influence of organizational events on the organizational structure of OSS projects, which could further be used to develop mechanisms to avoid critical consequences of organizational events with negative connotation.

1.2 Contributions

The contributions of this thesis are multifaceted. In what follows, we summarize them independently for our three research objectives. Figure 1.1 provides an overview of our different contributions and how they are related to each other.

Insights into the Evolution of Collaboration and Communication To address RO$_1$, we perform an in-depth analysis of developer interaction in OSS projects, using different analysis methods. This includes an investigation of synchronous development at different abstraction levels of coordination, to find out whether developers collaborate and communicate temporally close by, and, if so, whether their communication is content-wise related to their joint coding activity. In particular, we provide a method for measuring the synchronization of activities and for raising the abstraction level of exploring synchronization between developers'

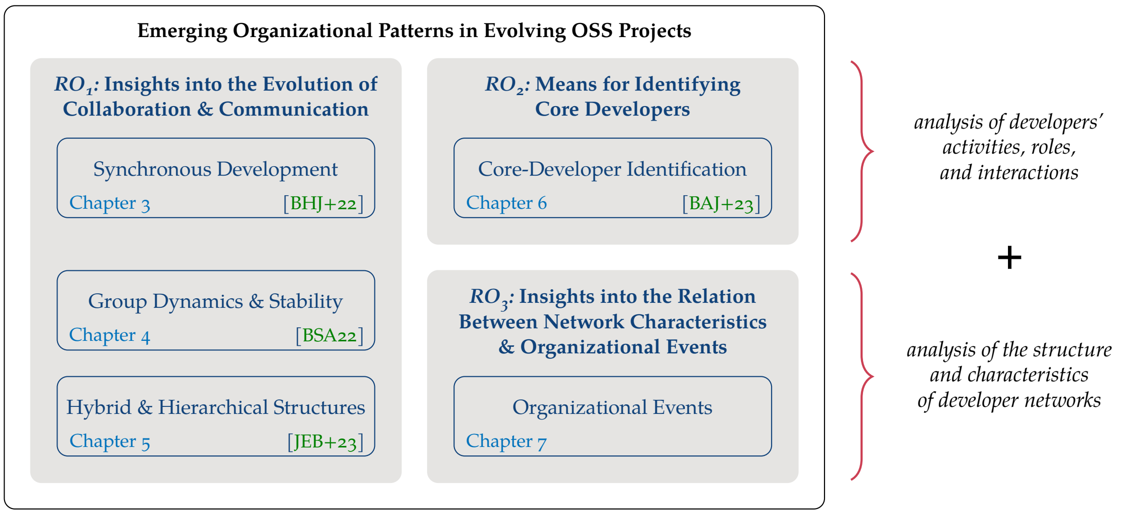

Emerging Organizational Patterns in Evolving OSS Projects
RO1: Insights into the Evolution of Collaboration & Communication
Synchronous Development
Chapter 3 [BHJ+22]
Group Dynamics & Stability
Chapter 4 [BSA22]
Hybrid & Hierarchical Structures
Chapter 5 [JEB+23]
RO2: Means for Identifying Core Developers
Core-Developer Identification
Chapter 6 [BAJ+23]
RO3: Insights into the Relation Between Network Characteristics & Organizational Events
Organizational Events
Chapter 7
analysis of developers' activities, roles, and interactions
+
analysis of the structure and characteristics of developer networks

Figure 1.1: Overview of our contributions and their interplay.

collaboration and communication activities to a conversation-based level. We introduce a continuous measure to quantify the synchronicity of co-editing source-code artifacts, and we provide insights into whether and to what extent developer communication is content-wise related to temporally close-by collaboration activities.

Furthermore, we use a multi-modal network perspective to examine how collaboration and coordination activities between developers can be explained by each other, whether the developers form stable group structures, and how these groups evolve over time. This information can be used for developing prediction models for future coordination, but also to detect instabilities in the organizational structure of an OSS project and search for potential causes of these instabilities. To that aim, we devise a modular, multi-step method to detect developer groups and provide a measurement and modeling framework that can be used to analyze group stability and to predict future interactions of developers. Using our framework, we are able to provide insights into the strength and stability of group structures in highly active OSS projects. While we are able to detect stable group structures in OSS projects, we are also able to identify distortions in the network structure. In this regard, our method provides the capability to analyze such distortions with respect to different interaction channels of developers and also with respect to different dynamic importances and developer roles.

Finally, we investigate the hierarchical structure of OSS projects over time to identify evolutionary patterns in the developers' communication. For that purpose, we develop a method to identify hybrid and hierarchical structures in developer networks, and we provide insights into the activities of developers with respect to their position in the hierarchical structure. In particular, we examine the distribution of tasks among the developers and the developers they interact with. This way, we obtain additional, general insights into the evolution of the organizational structure of OSS projects.

Means for Identifying Core Developers To address RO_2, we provide an overview of the core-developer identification methods that have been used in the literature, and we devise a method to automatically identify core developers based on the privileges they have in GitHub issues and pull requests. We validate this method with officially documented lists of core developers, and we compare the outcomes of this method to the results of state-of-the-art unsupervised developer-classification methods. Our results indicate that the choice of the interaction channel that is used for developer-role classification matters more than the actual classification metric. Furthermore, we provide perspectives on how our automatic method to identify core developers can be used to develop supervised classification methods.

Insights into the Relation Between Network Characteristics and Organizational Events To address RO_3, we investigate different organizational events in popular OSS projects, such as the emergence of project forks, the investment of a company, or the abandonment of an important core developer. In particular, we analyze whether and how these events have influenced the organizational structure of the projects and the characteristics of the corresponding developer networks. By means of an explorative study, we obtain insights into the evolution of OSS projects. Notably, we observe different changes in different network characteristics with respect to different organizational events. Based on our observations, we develop hypotheses that shall guide future research on the relationship between organizational events and the characteristics of developer networks. Thus, with our investigations, we lay foundations for the future development of means to detect structural distortions and to devise countermeasures that could be applied before a project takes harm of particular events.

1.3 Outline

This thesis is structured as follows: In Chapter 2, we provide the necessary background on OSS projects and their different interaction channels, on different roles in OSS projects, and on the construction and analysis of socio-technical developer networks.

To address RO_1, we conduct three different studies: In Chapter 3, we describe our empirical study on synchronous development in OSS projects. In Chapter 4, we investigate group dynamics and group stability in OSS projects. In Chapter 5, we look at hierarchical organizational structures of OSS projects.

To address RO_2, in Chapter 6, we provide an overview of how core developers have been identified in the literature, we devise an automatic method to identify core developers based on GitHub issues and pull requests, and we conduct an empirical study to validate our method and assess the accuracy of state-of-the-art developer-classification methods.

To address RO_3, in Chapter 7, we investigate the relationship between organizational events and changes in developer-network characteristics of OSS projects.

Finally, in Chapter 8, we summarize our findings and provide perspectives on future work that is based on our contributions. Appendix A contains links to supplementary materials (e.g., data, scripts, results, visualizations) related to all the empirical studies that we present.

Background 2

This chapter shares material with Bock et al. [BAJ+23; BHJ+22] and Joblin et al. [JEB+23].

In this chapter, we provide the background information and definitions that are relevant for several chapters of this thesis. First, we describe how developers coordinate in OSS projects and define the terms collaboration, communication, and coordination. Then, we characterize the different roles developers can take in OSS projects based on the literature. Afterwards, we introduce the concept of developer networks and the techniques and metrics that we use to analyze them. Last, we elaborate on how we extract data from OSS projects and describe the toolchain that we use to process the data.

2.1 Developer Coordination in Open-Source Software Projects

Open-source software (OSS) projects are software projects in which the source code is publicly available and licensed in such a way that any person can use, change, and redistribute it.[1]

In large OSS projects, often numerous globally distributed and independent developers contribute [Her07]. While this demonstrates the openness of OSS projects toward new developers and helps attracting new developers, this could also be problematic with respect to coordination issues. For example, when multiple developers contribute to interrelated source-code fragments, changes that lack coordination can introduce unintentional side effects. Consequently, developers must coordinate their interdependent activities to prevent conflicting changes, to avoid bugs, and to keep the code simple and maintainable [Bir11; CHC08; CMR+09; KSD11]. Hence, coordination is one of the key factors toward project success. Previous work has shown that both, collaboration and communication, are necessary.

2.1.1 Collaboration, Communication, and Coordination

The terms collaboration, communication, and coordination are widely used in the literature on studying socio-technical aspects of software projects [e.g., Bir11; CH08; CH13; CWH+06;

1 The Open Source Initiative provides a formal definition of "open source" and clearly states the requirements for the source code and its license to be considered as "open source": https://opensource.org/osd/ (accessed at 2023-05-31). In the context of this thesis, however, we consider projects to be "open source" when the source code of the project is publicly available and everyone can contribute to the project, while we neglect all the other licensing requirements, for the sake of simplicity, as the concrete license does not matter for investigating developers' coordination in such projects.

EKL+01; GHP99; HM03b; HMR06; HSA20; JAM17; KS95; KSD11]. Nevertheless, as there are different interpretations of these terms, let us define a precise meaning of these terms, which we use throughout this thesis:

Definition: Collaboration

Collaboration means that two developers work together by contributing to (i.e., co-editing), at least, one common source-code artifact. A source-code artifact can be a file or function, for instance.

Definition: Communication

Communication means that two developers talk to one another, also in written form (e.g., exchanging e-mails on a mailing list, or commenting on the same issue of an issue tracker).

Note that we define collaboration and communication each as an interaction between two individual developers. Nevertheless, in OSS projects, where source code and discussion platforms are publicly accessible, also more than two developers can interact with each other, of course, either via collaboration or via communication. Yet, we treat such cases as pairwise interactions between all the developers who collaborate or communicate with each other, respectively, in accordance with our definitions that we have given above.

Based on literature from organizational science, Malone and Crowston [MC94] define coordination as "managing dependencies between activities". In accordance with that, since there are communication activities that are related to developers' collaboration activities, we define the term coordination as follows:

Definition: Coordination

Coordination means that two developers are collaborating and communicating in (content-wise related) temporally aligned manners.

That is, coordination captures that developers communicate temporally close-by to their collaboration activities. Ideally, but not necessarily, in coordination, communication covers a topic that is related to what the developers are collaborating on [Bir11; HM03b; HMR06; KS95; KSD11]. This is in line with general definitions of coordination from sociology, which define coordination as "integrating or linking together different parts [...] to accomplish a collective set of tasks" [VDK76]. For example, when two developers edit the same file within a couple of days (i.e., they collaborate) and when they also exchange e-mails on the mailing list (i.e., they communicate) to discuss their contemporary edits to this file, we consider this as coordination, since collaboration and communication activities are temporally aligned and content-wise related. In Figure 2.1, we provide some examples that visualize the terms collaboration, communication, and coordination.

2.1.2 Coordination Requirements and Socio-Technical Congruence

According to Cataldo et al. [CWH+06], there are *coordination requirements* in OSS projects, which determine "who must coordinate with whom to get the work done." [CWH+06]. Consequently, they stated that fulfilling these coordination requirements via coordination

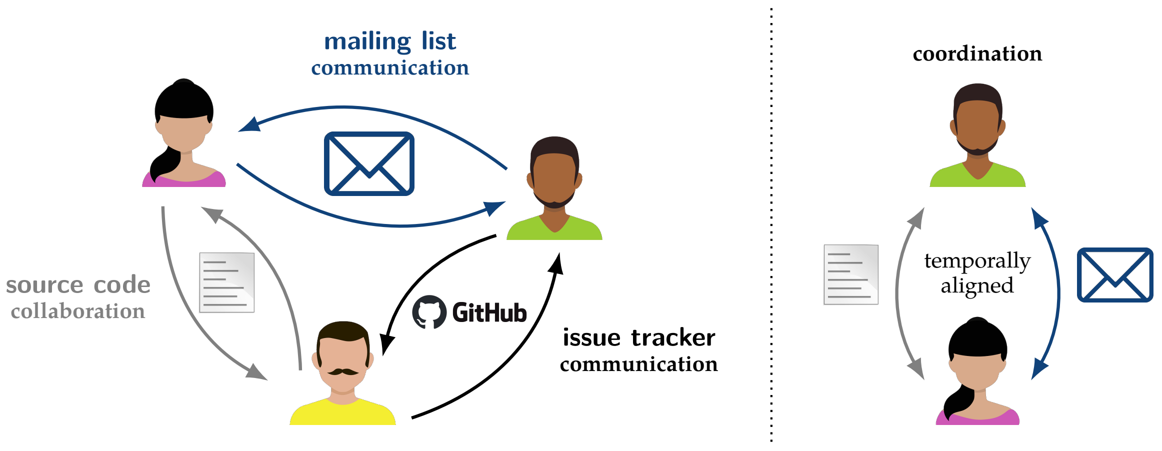


Figure 2.1: Overview of different interaction channels in OSS projects: (left) collaboration on the source code, and communication on issue tracker and mailing list; (right) coordination between two developers who collaborate and communicate temporally close-by.

activities reduces the development time and increases developer productivity. There is a substantial corpus of research in which coordination requirements are analyzed, which conclude that coordination requirements emerge when two developers work on the same source-code artifacts, such as the same file, to reduce the risk of introducing dysfunctional source code and to resolve interrelated dependencies [CWH+06; HM03a; HMR06; HSA20; MC90; XGD+12]. As coordination captures collaboration and communication activities that are related to each other, Cataldo and Herbsleb [CH13] and Cataldo et al. [CHC08; CWH+06] coined the term *socio-technical congruence*. That is, to fulfill the coordination requirements, the technical activities (i.e., collaboration) and the social activities (i.e., communication) in the project should be in accordance with each other. In this context, "Conway's Law" [Con68] states that the architecture and design of software follows the organizational structures of the communication activities. Even more, according to the "mirroring hypothesis" [CB16; KCD12], organizational structures in technical and social activities should align due to their common tasks. Noteworthy, the relationship between technical and social activities in OSS projects has been analyzed from many different perspectives in an extensive body of empirical research [e.g., BGK+13; BMF+13; Bir11; HG99; HSA20; Kam19; MJT+22; RAN+21; SH13; VHE+07].

2.1.3 Interaction Channels in Open-Source Software Projects

When developers perform technical or social activities, in OSS projects, they usually interact with other developers (be it intentionally or unintentionally, due to the nature of OSS projects). To interact with each other, there are usually different interaction channels in OSS projects, which cover different types of interactions:

Collaboration Channels On the one hand, there is the *version-control system* (*VCS*) of a project, in which the source code of the project is managed. The VCS, in which the source code of the project is stored, often is also referred to as *repository*. In particular, developers can access the source code from the VCS, modify the source code, and submit the modified version of the source code back to the VCS [AKM08; DP03; Som10]. Such source-code modifications can implement bug fixes, refactorings, or further enhancements of the software, for instance. The VCS stores all the source-code modifications as uniquely identifiable *commits*, which are "atomic changes done on the source code" [GM10]. In a commit, the particular source-code modifications are stored together with metadata such as author information (name and e-mail address) or modification timestamps [AKM08]. The most frequently used VCS today is Git[2] [DPK+20], which is a distributed system that allows the developers to work on their local clones of the repository offline as well as synchronizing their changes with other developers' changes in a remote repository [Spi12]. Beside Git, there are also other VCSs: For instance, the Mozilla Foundation uses Mercurial[3] as VCS for the development of their browser Firefox,[4] Canonical uses the VCS Bazaar[5] to maintain Ubuntu packages,[6] and the Apache Software Foundation uses the VCS Subversion (SVN)[7] for the development of their web server Tomcat.[8] Due to the prevalent usage of Git in today's software development, in this thesis, we only analyze projects that use Git as their VCS. Yet, other VCSs behave very similarly to Git in how they store developers' commits [KT11; ZND18], which is why the methods and analyses that are proposed in this thesis are easily transferable to projects that use other VCSs.

Communication Channels On the other hand, there can be multiple communication channels in an OSS project, in which the source-code modifications and other project-related topics are discussed [EGW22; KGB+18; SSF+17; TNK+19]. In this thesis, we mainly consider two established communication channels: mailing lists and issue trackers.

Mailing lists are usually used by OSS projects that date back more than 10 years and, therefore, are a historically rich and well-established source of communication data [RGS08; RLM19]. On the developer mailing lists of OSS projects, usually software architecture and source-code modifications are discussed. Particularly, numerous projects have policies[9] that force developers to send source-code patches (i.e., drafts of commits) to the mailing list for discussion and code review before the patches are committed to the VCS [RLM19]. A recent study has shown that about 89% of code-review discussions in such projects take place on the mailing list [MAJ+20]. In addition, mailing lists also contain discussions of the outcomes of developer conferences (or similar events) where complex issues and long-term plans for the future development of the project are addressed. Even developers in OSS projects who work for corporations may use mailing-list discussions to communicate their intentions to others, as public communication is one of the basic concepts in OSS projects [Rie15].

Issue trackers are more recently used communication channels that are dedicated to report and discuss project-related issues on specialized, software-development specific web interfaces [BLJ+13]. During the last decade, social coding platforms such as GitHub[10] or Gerrit[11]

2 https://git-scm.com/ (accessed at 2023-06-08)
3 https://www.mercurial-scm.org/ (accessed at 2023-06-08)
4 https://firefox-source-docs.mozilla.org/contributing/vcs/mercurial.html (accessed at 2023-06-08)
5 https://bazaar.canonical.com/ (accessed at 2023-06-08)
6 https://wiki.ubuntu.com/UbuntuDevelopment#Revision_control_.28Bazaar.29 (accessed at 2023-06-08)
7 https://subversion.apache.org/ (accessed at 2023-06-08)
8 https://people.apache.org/~pidster/tomcat/site-2011/svn.html (accessed at 2023-06-08)
9 As an example, we refer to the contribution policy of the project QEMU:
https://www.qemu.org/docs/master/devel/submitting-a-patch.html (accessed at 2023-06-08)
10 https://github.com/ (accessed at 2023-06-08)
11 https://www.gerritcodereview.com/ (accessed at 2023-06-08)

have established [BB13; DST+12; PT22]. These platforms host the source code of a project in the VCS together with dedicated communication channels (e.g., issue trackers or code reviews), in which discussions can be directly linked to source code and commits, and vice versa. To this end, GitHub provides issues[12] and pull requests (PRs)[13] as communication channels for developers. Whereas issues are mainly used to discuss potential bugs or request new features, pull requests are used to review source-code modifications before they are finally integrated into the project's code base [RR14]. Note that GitHub internally treats pull requests as a special form of issues.[14] For that reason, in this thesis, when we talk about issues this always also includes pull requests. As already discussed for VCSs above, GitHub issues are not the only used means to track issues and bugs. Many projects use separate issue trackers such as Bugzilla[15] (e.g., Mozilla uses Bugzilla to track bugs in Firefox[16]) or Jira[17] (e.g., the Apache Software Foundation uses Jira to track issues in their projects[18]) [HZ13; MLM22]. Due to the prevalence of GitHub, we stick to the analysis of projects that use GitHub issues as their issue tracker when analyzing issue trackers, in this thesis. Yet, the processes of reporting and discussing bugs and issues are similar in other issue trackers [DML16; QST22], which is why we assume that our methods and results are transferable to other social coding platforms as their purpose and usage scenarios are equivalent.

Among others, there are also additional communication channels such as direct communication between developers (e.g., sending personal e-mails, using chat systems, or online forums) [EGW22; KGB+18; SSF+17]. Whereas some of these channels are not publicly accessible (e.g., personal e-mails), others may be available for the public (e.g, online forums). However, in this thesis, we analyze only mailing lists and GitHub issues, as they are a widely used and publicly available data source of developers' communication that, in addition, is directly related to their source-code modifications [GBL+13; PT22; RR14; SSF+17]. That is, in projects that use mailing lists, source-code modifications are discussed on the mailing list before they are integrated into the code base [RLM19], and, in projects that use GitHub issues, commits are reviewed and discussed in pull requests prior to being merged into the main branch [RR14]. We chose to cover mailing lists and GitHub issues to analyze two different communication channels that are used in many projects and that are related to the source-code modifications, to obtain a more holistic view on communication and coordination in OSS projects. In Section 2.4, we describe how we extract the corresponding data from the different sources of information.

Altogether, in this thesis, we consider the VCS as a collaboration channel, and we consider issues as well as mailing lists as communication channels. To provide an overview of all the different developer interaction channels that we cover in this thesis, we have depicted them in the left part of Figure 2.1 (page 9).

12 `https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues/` (accessed at 2023-06-08)
13 `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests/` (accessed at 2023-06-08)
14 `https://docs.github.com/en/rest/issues/issues?apiVersion=2022-11-28/` (accessed at 2023-06-29)
15 `https://www.bugzilla.org/` (accessed at 2023-06-08)
16 `https://www.bugzilla.org/contributing/reporting_bugs.html` (accessed at 2023-06-08)
17 `https://www.atlassian.com/software/jira/` (accessed at 2023-06-08)
18 `https://issues.apache.org/jira/` (accessed at 2023-06-08)

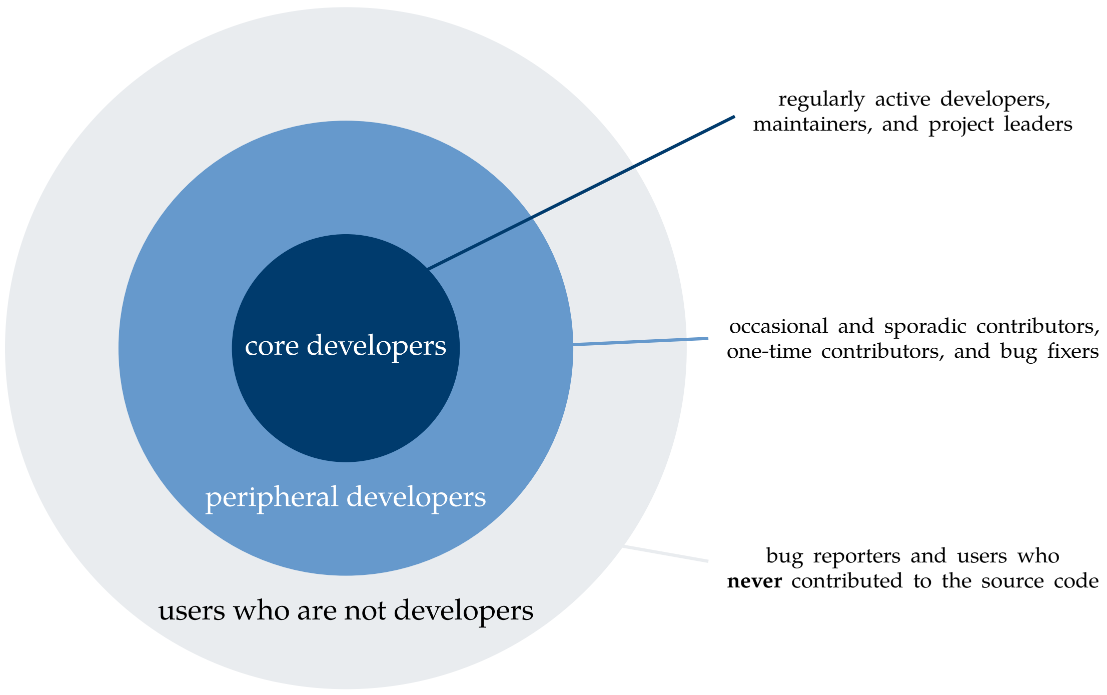

regularly active developers, maintainers, and project leaders
core developers
occasional and sporadic contributors, one-time contributors, and bug fixers
peripheral developers
bug reporters and users who **never** contributed to the source code
users who are not developers

Figure 2.2: Overview of the different developer roles that we use in this thesis.

2.2 Developer Roles in Open-Source Software Projects

When developers contribute to software projects (either via contributions to the source code or via discussions on the communication channels), they undertake different kinds of tasks. This especially holds for OSS projects, in which volunteers can participate and in which developers are globally distributed [GHP99; Rie15; XJS09]. For instance, some developers perform maintenance tasks and are involved in the project in a long-term manner, whereas other developers participate in the project only to implement a specific feature. Consequently, developers in OSS projects have different roles.

Nakakoji et al. [NYN+02] proposed the so-called "onion model" distinguishing eight different roles of community members: There are several roles for project users (passive user, reader, bug reporter), who do not contribute to the project's source code. Five roles are directly related to source-code contributions: bug fixer, peripheral developer, active developer, core member (also called maintainer), and project leader (who are mostly the project initiators). Xu et al. [XGC+05] proposed a similar model composed of four developer roles. As the distinction into four or five different developer roles is rather fine-grained and their boundaries are blurred, researchers often conflate them into just two roles: Whereas bug fixers and peripheral developers only contribute occasionally and sporadic, according to the "onion model", active developers, core members, and project leaders contribute regularly. As a consequence, the occasionally and sporadically contributing developers are called *peripheral* developers, whereas the remaining ones are called *core* developers or *maintainers* [CH05; CWL+06; JAH+17; JAM17; MFH02; TRC10]. Researchers sometimes additionally consider the role of one-time contributors (also called newcomers), who contribute only once to a project (e.g., provide a single bug fix) [CZS+20; LC17]. One-time contributors, though, are considered to be part of the group of peripheral developers [LCB17; PSG16; SRS+12]. In Figure 2.2, we provide an overview of the developer roles between which we distinguish in this thesis.

In addition to the above mentioned developer roles, there are also other approaches on how developer roles can be defined. For example, Cheng et al. [CLL+17] distinguish between developmental core developers and collaborative core developers. However, such a distinction is not disjoint, as highly active developers may also collaborate with many others. Constantino et al. [CZS+20] differentiate between project roles and committer roles. Whereas the project roles represent various organizational and potentially overlapping tasks (developer, maintainer, team leader, project promoter, reviewer, or coordinator), the committer roles encompass what we consider as core and peripheral developer roles. Montandon et al. [MVS21] distinguish technical roles of GitHub users across projects, driven by the variety of technical tasks a developer mostly takes across all the projects a developer contributes to (e.g., contribute to the frontend, to the backend, or to the continuous integration), whereas we study the activity role of a developer within a specific project.

Furthermore, Trinkenreich et al. [TGW+20] found, by means of an interview study, that beside "project-centric roles" (such as programming or system administration), in OSS projects, there are also "community-centric roles" that are not related to programming tasks at all, such as strategic managing tasks "to make projects more welcoming, inclusive, and sustainable". As such management tasks are often not visible in the source-code repositories, contributors that take on such roles are also called "hidden figures" [TGW+20]. While these management tasks are important for the sustainability of an OSS project, in this thesis, we mainly focus on developers' activities that are related to source-code changes and investigate how they are reflected in communication activities.

2.3 Analyzing Developer Networks

To detect developer roles or to obtain insights into the organizational structure of an OSS project, researchers often use a network perspective of the project [e.g., AW18; BC14; BGD+06; CWL+06; DKS+10; EKB+17; GLM06; HAT+21; JAH+17; JAM17; LRG+06; MGJL21; MW11; PT21; TPS+19]. Previous work has shown that adopting methods and concepts from social-network analysis is also beneficial for a socio-technical analysis of OSS projects. A network perspective enables researchers and project stakeholders to model and visualize the relationships and connections between different developers and their activities in the project. For example, if a developer is connected to many other developers, this could be an indicator that this developer should be treated to be a core developer. Besides, the network perspective also allows to measure different network characteristics and their evolution over time, which can be valuable information to detect changes in the organizational structure of an OSS project. In the following, we introduce the different types of networks that we use in this thesis, as well as the network characteristics and network metrics that we employ in our analyses.

2.3.1 Building Developer Networks

We model a *developer network* as a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges [BE05; Ste10]. In our setting, the set of vertices consists of all developers who have participated in the project, and the set of edges consists of the connections between the developers. That is, an edge $e = (d_1, d_2) \in E$ with $d_1, d_2 \in V$ exists if there is a connection between developers d_1 and d_2. These connections can model different relationships between the developers, which is why we differentiate between different *network types*:

cochange: In a cochange network, an edge between two developers represents the collaboration between these two developers. That is, the developers have both edited a common source-code artifact (e.g., file or function) [GHJ98; JMA+15; JSS11; PBD+14; PD08; ZZW+05].

mail: In a mail network, an edge between two developers represents the communication between these two developers on the mailing list. That is, the developers both sent an e-mail to the same e-mail thread [BGD+06; BPD+08; PBD+14; SFD05; SSS17].

issue: In an issue network, an edge between two developers represents the communication between these two developers on the issue tracker. That is, the developers both contributed to the same issue, for instance, by commenting on or reviewing the issue [CWL+06; HKC+11; MJT+22; OHM+18; PBD+14; SGR11].

In addition, we build combined network types that consist of multiple of the above described relationships. For instance, the *cochange+issue* network consists of edges that represent the collaboration between two developers on the source code and of edges that represent the communication between two developers on the issue tracker. In such combined networks, we keep the information which edge has originated from which individual network type. However, in this thesis, we do not make use of this information and, thus, ignore the type of an edge in combined networks, as we analyze the combined networks only in addition to the individual network types. Using combined network types, we can analyze multiple interaction channels together and can get a more general view on how the developers interact with each other, may it be collaboration or communication, independent of the concrete channel.

To emphasize the underlying data source and the activities the developers perform, we sometimes refer to cochange networks as *collaboration networks* and to mail or issue networks as *communication networks*.

When we consider only a subset of the vertices V of a network $G = (V, E)$ (i.e., a subset of the developers) and the edges between the vertices of the subset, the resulting network is called a subgraph or a *sub-network* [Ste10].

During the construction of networks, one needs to decide on two network properties, namely *directedness* and *simplicity*, which both affect the way how edges are constructed.

Directedness If a network is *undirected*, its edges do not have a direction. That is, $(d_1, d_2) = (d_2, d_1)$. So, the edges (d_1, d_2) and (d_2, d_1) are considered to be identical as both model a relationship between d_1 and d_2. Sometimes, this is also denoted by $\{d_1, d_2\} \in E$ to show that the order of the vertices when defining an edge is arbitrary in undirected networks [BE05]. In a *directed* network, however, $(d_1, d_2) \neq (d_2, d_1)$ holds, which means that it matters who has initiated the relationship [BE05]. In our developer networks, if we use directed networks, the direction of an edge describes a temporal order of two events. For example, if developer d_1 replies to an e-mail of developer d_2, then d_1 initiates the relationship between them, and, therefore, we model their relationship by an edge from d_1 to d_2. Similarly, when developer d_1 changes a file that has been changed by developer d_2 beforehand, we model this relationship in the developer network as an edge from d_1 to d_2. In Figure 2.3, we visualize an example of a

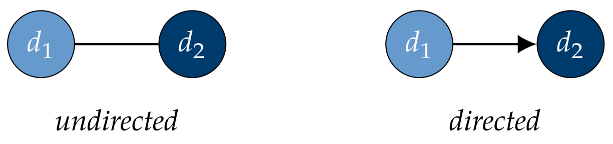


Figure 2.3: Directed and undirected developer networks: (left) vertices d_1 and d_2 are connected by an undirected edge; (right) vertices d_1 and d_2 are connected by a directed edge, that is, d_1 is the initiator of the interaction (e.g., d_1 changes a file that d_2 has already changed beforehand).

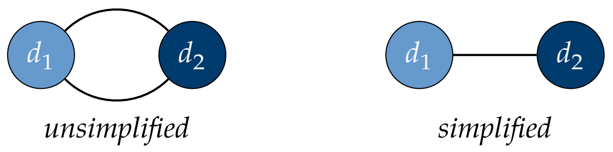


Figure 2.4: Simplified and unsimplified developer networks: (left) vertices d_1 and d_2 are connected by multiple edges and, thus, the network is unsimplified (by definition); (right) vertices d_1 and d_2 are connected by a simplified edge.

directed and an example of an undirected network. When modeling developers' interactions, the choice of using directed or undirected networks depends on whether the temporal order between two events should be considered or not. However, it is worth to note that, in many cases, and also for many network metrics that we will introduce below, the direction of edges is ignored even when the network is directed.

Simplicity Beside the direction, one also needs to decide whether the frequency of interactions between two developers matters, or whether just the fact that there is an interaction between the two developers at all should be modeled, no matter to which extent. In the latter case, in which only one edge between any two developer is allowed, the network is called *simplified*, since multiple edges become simplified to a single edge. In this case, the original number of edges between these two developers can be stored as an additional edge attribute, called *edge weight*, if necessary. In simplified networks, also loop edges (i.e., edges from a vertex to itself) are removed, as such edges do not model an interaction between two vertices [Ste10]. Opposed to that, when we allow a network to have multiple edges between a pair of vertices or to contain loops, the network is called an *unsimplified* network. In Figure 2.4, we provide an example of a simplified and an example of an unsimplified network. In general, simplification matters when the number of connections from one vertex to another is analyzed: It makes a difference whether a developer has 100 connections to 100 different developers or 100 connections to the very same developer. In an unsimplified network, we cannot differentiate between these two cases by simply counting edges, whereas, in a simplified network, the latter 100 connections would be conflated into one single edge. Thus, using simplified networks helps avoid that a network is distorted by a pair of developers that has extraordinarily many activities (e.g., many replies to each other in a single issue or mailing-list thread) that would lead to an extraordinary high number of edges. However, by simplification, we also lose information about the number of interactions (e.g., communication events), which could be a useful piece of information when assessing developers' roles based on their activities in a project.

In this thesis, we cover both, simplified and unsimplified, as well as directed and undirected networks, as all four combinations of these properties provide different views on the network structure and allow us to abstract from OSS projects in different ways.

2.3.2 Subsequent and Sliding Time Windows

When analyzing the collaboration and communication of large and long-lasting OSS projects, it is important to think about the time period that is used to build a developer network. For example, if two developers have edited the same file but with a temporal distance of 20 years, the therefrom resulting edge might not really capture a collaboration activity between these developers. This is why we expect developers' activities to be temporally close-by in order to be treated as collaboration (as already mentioned above in Section 2.1). To account for temporal proximity of developers' activities when constructing developer networks, we split networks into *subsequent windows* that capture a specific time period of length l, such as 6 months, for example, as common in the literature [e.g., HKC+11; MW11; PBD+14]. More formally, let $T = [t_0, t_{\max}]$ be the whole time period of a project that we analyze (for which we were able to extract collaboration and communication data), where t_0 denotes the earliest point in time for which we have data (e.g., the first commit or the first e-mail), and $t_{\max}$ denotes the latest point in time (e.g., the last commit or e-mail that we analyze in our dataset). Then, we get the set of subsequent time windows of length l (in months), $T_{l,\,\text{subsequent}}$, as follows:[19]

$$T_{l,\,\text{subsequent}} = \Big\{ \big[(t_0 + x \cdot l),\ (t_0 + (x+1) \cdot l) \big[\ \Big|\ x \in \{0, 1, \dots, n\}, \\ t_0 + (n+1) \cdot l \leq t_{max},\ n \in \mathbb{N} \Big\} \tag{2.1}$$

We refer to the elements of $T_{l,\,\text{subsequent}}$ as *time windows* or *time ranges*. For example, when l = 6 months, we also talk about *6-months ranges*. When we build developer networks, we build a separate network for each of the time ranges that are contained in $T_{l,\,\text{subsequent}}$. This way, we can make sure that the activities that cause an edge in the network have a maximal temporal distance of l. However, using this condition for the edge construction, we miss edges that cross the borders of two subsequent time ranges. Therefore, we sometimes also use *sliding windows*, in which we add overlapping time ranges that capture half of the preceding and half of the succeeding time window, not to miss the edges that cross the time windows [JAM17]. Accordingly, the set of sliding windows of length l, $T_{l,\,\text{sliding}}$, is defined as follows:

$$T_{l,\,\text{sliding}} = T_{l,\,\text{subsequent}} \cup \\ \Big\{ \big[(t_0 + \frac{1}{2} l + x \cdot l),\ (t_0 + \frac{1}{2} l + (x+1) \cdot l) \big[\ \Big|\ x \in \{0, 1, \dots, n'\}, \\ t_0 + \frac{1}{2} l + (n'+1) \cdot l \leq t_{max}, \\ n' \in \mathbb{N} \Big\} \tag{2.2}$$

19 We consider 0 to be part of $\mathbb{N}$. This holds throughout the whole thesis.

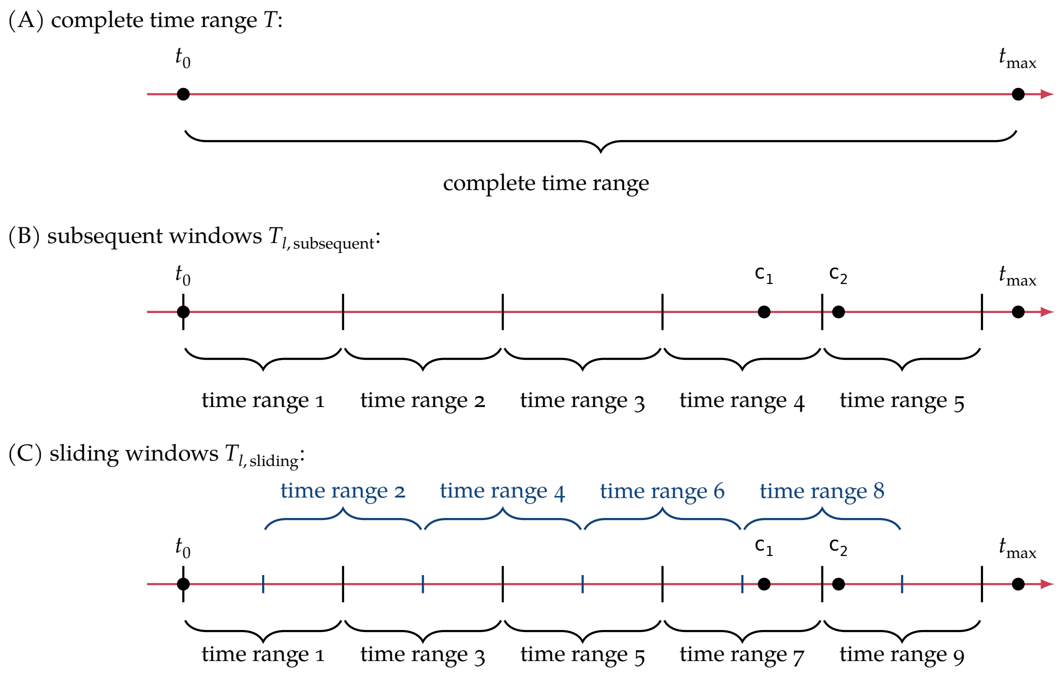


Figure 2.5: Assume that t_0 is the initial commit of a software project and t_{max} represents today. (A) The complete time range of all available data is covered by the interval $[t_0, t_{\text{max}}]$. (B) When we use *subsequent* time windows, in this example, we get 5 time ranges of equal length l between t_0 and t_{max}, namely the time ranges 1, 2, 3, 4, and 5. If there is a relationship between activity c_1 and activity c_2, we are *not* able to cover this relationship as an edge in any network, as there is no time range that contains both activities. (C) When we use *sliding* windows, we end up in 9 overlapping time ranges of the same length l, where the time ranges 2, 4, 6, and 8 cover the second half of their preceding and the first half of their succeeding time range. If there is a relationship between the activities c_1 and c_2, we are able to relate them to each other by an edge in the network of time range 8, as this time range contains both activities.

Again, we refer to the elements of $T_{l,\text{sliding}}$ as *time windows* or *time ranges* without explicitly mentioning the sliding windows if it is mentioned beforehand that, in a specific study, sliding windows are used to construct the ranges.

In Figure 2.5, we exemplarily visualize T, $T_{l,\text{subsequent}}$, and $T_{l,\text{sliding}}$. Moreover, in this figure, we demonstrate that we might miss some edges between two activities that cross time ranges when we use subsequent time ranges, while we are able to cover these edges when we use sliding windows. Note that the last time range may end earlier than t_{max} (see also the corresponding $\leq$ conditions in Equations 2.1 and 2.2). That is, if the time period between the end of the last time range and t_{max} does not sum up to a full time range of length l any more, we ignore this time period and do not take it into account for our analyses, as considering this time period would distort our results due to considering incomplete time ranges or time ranges of unequal lengths.

2.3.3 Characteristics of Networks

When analyzing networks, it may be worth looking for specific characteristics of these networks. In particular, when analyzing the evolution of networks, specific characteristics of these networks and their changes can be used to reveal how the networks change over time. Especially in the context of developer networks, their characteristics can provide valuable information about the organizational structure of the underlying OSS project, about the roles of the individual developers in the project, and also about the alignment of collaboration and communication in the project [e.g., AW18; EKB+17; HSA20; JAH+17; JAM17; ZYW11].

When analyzing network characteristics, we differentiate between characteristics that cover the whole network (i.e., the whole project, within a given time range) and characteristics that are specific for individual vertices (i.e., developers in the networks that we analyze). Let us begin with the characteristics of the whole network, which describe the structure and connectedness of a network in its entirety:

Average Degree The *degree* of a vertex v is the number of edges a vertex has [Bar16; Ste10]. That is, in an undirected, simplified network, the vertex degree represents the number of vertices to which a vertex is connected. Remember that, in a directed or unsimplified network, there can be more than one edge between a pair of vertices. While the vertex degree is computed separately for each vertex, the *average degree* is a network characteristic that describes how many edges a vertex in the network has on average [Bar16]. Thus, the average degree is a rough indicator for the number of interactions an average developer is involved in.

Density When analyzing networks, the number of existing edges in the network plays a vital role. Therefore, the *density* of a network is defined to be the ratio of the number of existing edges compared to the maximum number of possible edges [WF94]:

$$\frac{\#\textit{existing edges}}{\#\textit{possible edges}} \tag{2.3}$$

Note that the density is only well-defined for simplified networks, in which there can be either no or exactly one edge (per direction in case of directed networks) between each pair of vertices. If every vertex is connected to every other vertex in the network, the density of the network is 1. However, if the network does not contain any edges, its density is 0. In a developer network, the density indicates the general level of developer interaction. Whereas a high density shows that many different developers interact with many others, a low density reveals that there are only few interactions between the developers [KG13].

Clustering Coefficients According to Brandes and Erlebach [BE05], clustering means putting the vertices of a network into "natural groups" [BE05]. So, the *clustering coefficient* is a metric that indicates how clustered, that is, how tightly connected specific parts of a network are. This information about a network can be valuable for assessing the developers' embeddedness in the project community and how interconnected the developers are (i.e., whether they interact with different developers that are not interacting with each other, or whether they interact with developers that are well connected with each other) [e.g., JAM17; LRG04; XCM06; ZG13].

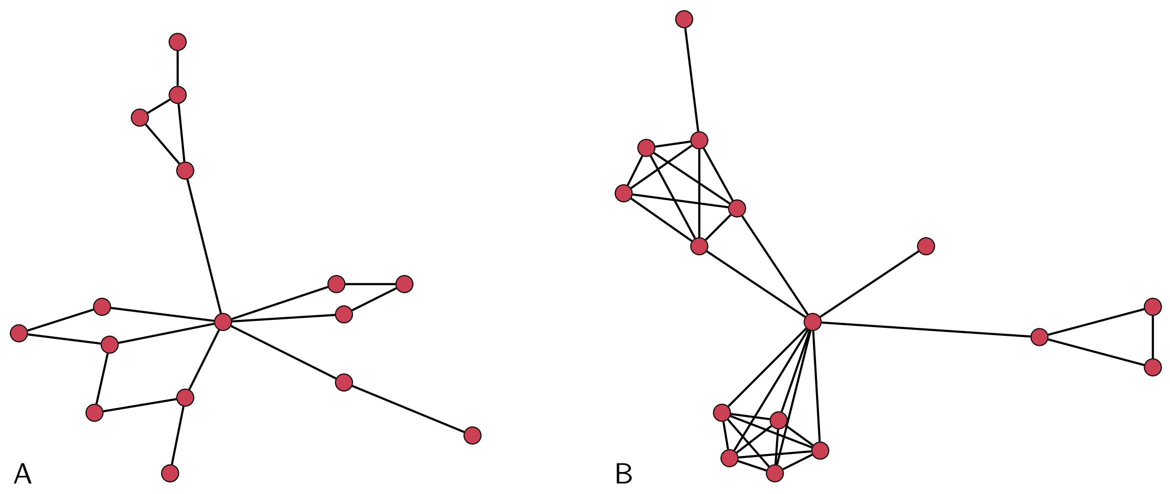


Figure 2.6: Networks A and B both have 16 vertices. The global clustering coefficient of network A is 0.07 because there is only one triangle in it, but it contains numerous connected triples. Opposed to that, network B consists of a lot of triangles and tightly-connected clusters, resulting in a global clustering coefficient of 0.72.

There are different means of how to compute clustering coefficients, for example, global or local clustering coefficients. Usually, the direction of edges is ignored while computing the clustering coefficient. The *global clustering coefficient* (sometimes called transitivity) can be calculated by relating the number of triangles in the network and the number of connected triples to each other [WF94]. For better understanding, a triangle in a graph G is a subgraph of G consisting of exactly three vertices that are pairwise connected [BE05]. Further, a connected triple is a set of three vertices that lie on a path of length two, that is, the first vertex is connected to the second one and the second one in turn to the third one. With that, the global clustering coefficient is calculated as follows [WF94]:

$$\frac{3 \times \#triangles}{\#connected\ triples} \tag{2.4}$$

The global clustering coefficient tends to be 1 if the considered network is tightly connected: Since each triangle can be considered as three different entities of connected triples, the numerator of the above fraction becomes equal to the denominator if the whole network consists almost entirely of triangles. In Figure 2.6, we show two example networks, to provide a visual perspective on what the global clustering coefficient describes. Whereas network A is weakly connected and, thus, results in a low global clustering coefficient of 0.07, network B is highly clustered and, therefore, has a comparably high global clustering coefficient of 0.72. As we can clearly see from the example network's visualization, the highly clustered network B contains multiple groups of developers (i.e., clusters) that are strongly interconnected. That is, within these tightly-connected groups, most of the developers directly interact with each other and, therefore, make a fast and easy exchange of information possible [GM07].

While the global clustering coefficient provides one number that indicates how the network is clustered, it does not allow any conclusions to be drawn about the role of the individual vertices in the network. The *local clustering coefficient*, however, is calculated for each vertex in the network, and it states how many neighbors of a vertex (i.e., vertices that are connected to

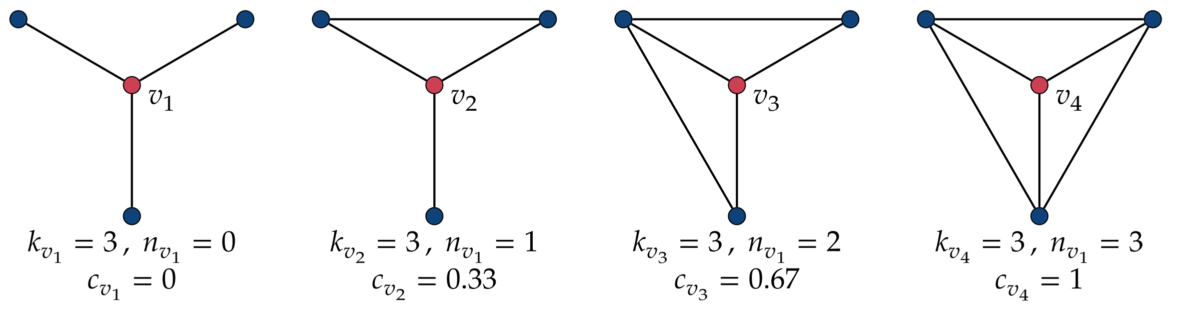

v_1
v_2
v_3
v_4
$k_{v_1} = 3,\ n_{v_1} = 0$
$c_{v_1} = 0$
$k_{v_2} = 3,\ n_{v_1} = 1$
$c_{v_2} = 0.33$
$k_{v_3} = 3,\ n_{v_1} = 2$
$c_{v_3} = 0.67$
$k_{v_4} = 3,\ n_{v_1} = 3$
$c_{v_4} = 1$

Figure 2.7: Four examples of the local clustering coefficient: c_{v_1} to c_{v_4} are calculated for four different vertices v_1 to v_4 according to the definition of the local clustering coefficient in Equation 2.5. If the neighbors of a vertex are not connected at all, the local clustering coefficient is 0. If the neighbors of a vertex are fully connected, the local clustering coefficient is 1.

the considered vertex) are connected with each other as well, compared to the maximum possible number of edges between these neighbors [WS98]. That is, the local clustering coefficient of a vertex is the ratio of the number of edges in the network that exist among all the neighbors of the considered vertex to the maximum number of edges that are possible among these neighbors. Thus, for a vertex i, the corresponding local clustering coefficient c_i is defined as follows:

$$c_i = \frac{2 \cdot n_i}{k_i \cdot (k_i - 1)}, \tag{2.5}$$

with n_i being the number of edges between the k_i neighbors of vertex i. Again, as for the global clustering coefficient, the local clustering coefficient c_i of a vertex i is 0 if none of its neighbors are connected and 1 if all of its neighbors are connected with all other neighbors. In Figure 2.7, we illustrate four examples of the values of the local clustering coefficient. As there is an individual local clustering coefficient for each vertex, this metric lacks a general view on the network as a whole. Nevertheless, to obtain a general metric for the whole network from the local clustering coefficients, they can be aggregated to form such a general metric. More specifically, to provide an example of such a general network metric, the *network average clustering coefficient* is the mean of the local clustering coefficients of all vertices in the network.

In general, there also exist many more different definitions of clustering coefficients than the ones we have presented above. They are not all equal, but the overall interpretation is almost the same for each definition of a clustering coefficient [BE05]. Yet, the different definitions of clustering coefficients focus on slightly different aspects of the connections in a network. In particular, there are other network characteristics (some of which we present below) that are determined based on the different definitions of clustering coefficients presented above.

Modularity When a network consists of multiple clusters (i.e., a group of vertices that are tightly connected), network scientists also call these clusters *communities* [Bar16]. According to Barabási [Bar16], "a community is a locally dense connected subgraph in a network". This definition is in line with the work of Newman and Girvan [NG04], who coined the term *community structure*, which they defined as a "division" of the network's vertices "into groups" for which the vertices within a group are densely connected with each other, but for which there are only sparse connections between the groups. To measure the quality of such divisions,[20] Newman and Girvan [NG04] defined a metric called *modularity*, which basically checks for each determined community whether the number of edges between the vertices that belong to the community is greater than the number of edges that would be expected from a random network with the same number of edges, the same number of vertices, and the same degree distribution (i.e., the distribution of the number of edges a vertex in the network has) [Bar16]. For a more thorough description of the computation of modularity and for a concrete mathematical definition, we refer to Clauset et al. [CNM04] and Newman and Girvan [NG04]. The higher the modularity, the better is the community structure in the network [Bar16]. Values above 0.3 indicate a "significant community structure" [CNM04]. Negative values indicate that the network lacks a community structure at all (i.e., each vertex is seen as a separate community) [Bar16].

Thus, a high modularity of a developer network would be an indicator for the existence of a community structure in the organizational structure of the software project. The existence of a significant community structure in a developer network reveals that individual developers form groups of developers who tightly interact with each other.

Average Path Length *Average path length* is a metric that computes the mean of the shortest path lengths between each of two pairs of vertices. In turn, a path between two vertices v_1 and v_s is a sequence of vertices and edges $v_1, e_1, v_2, e_2, ..., e_s, v_s$ with $e_i = (v_i, v_i + 1) \in E, v_i \in V, i \in \mathbb{N}$, in which an edge e_i appears, at most, once [BE05]. The length of a path is the number of edges within the path. Hence, the shortest path between two vertices is the path between them with minimal number of edges. In directed networks, both directions between the two vertices are considered to calculate the average path length. In general, the average path length represents the average number of vertices that are necessary to reach any other vertex in the network. In developer networks, it indicates the average number of developers that need to pairwisely interact with each other to pass information from one developer to any another developer in the network. Instead of interpreting the average path length for developers directly on its own, we use average path length in combination with other metrics to obtain more advanced characteristics of the structure of a developer network, such as small-worldness, which we describe next.

Small-Worldness In the literature, not only numeric metrics (such as clustering coefficient or average path length) are used to characterize networks and their structure, but also classes of different networks have been defined. According to Watts and Strogatz [WS98], there exist three different classes of networks: regular networks, small-world networks, and random networks. One central characteristic of *regular networks* is a high (local) clustering coefficient. *Random networks*, on the other hand, do have small average shortest path lengths. As *small-world networks* lie between regular and random networks, they have both characteristics, namely high clustering coefficient and small average path length [BE05].

Humphries and Gurney [HG08] developed a technique to express *small-worldness* in terms of numbers. According to their approach, the network to analyze is compared to an Erdős-Rényi (ER) random network [ER59] of the same size, that is, a random network

20 To identify such divisions, there are many different algorithms for *community detection*, which might lead to detecting different communities. For the sake of simplicity, in this thesis, when we refer to modularity, we simply rely on the walk-trap community-detection algorithm, which is well-established [SZF+20; YAT16] and traverses the network by randomly walking along its edges. For more details on that, we refer to Pons and Latapy [PL05].

having exactly the same number of vertices and the same number of edges as the network to analyze. To do so, the (global) clustering coefficients C and the average path lengths L of both the random network and the network to analyze are computed. With that, they build the following fraction S, which is a fraction of two other fractions:

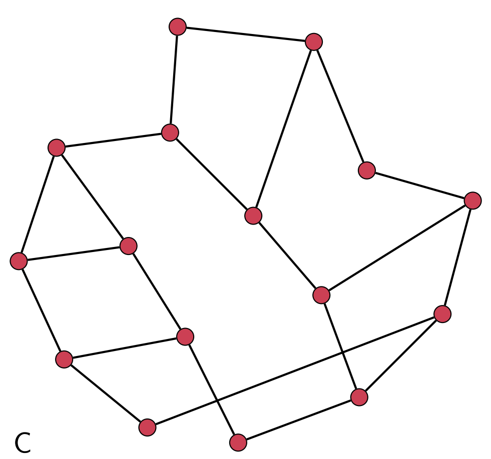
$$S = \frac{\gamma}{\lambda}, \quad \text{with} \quad \gamma = \frac{C_{analysis}}{C_{random}} \quad \text{and} \quad \lambda = \frac{L_{analysis}}{L_{random}} \tag{2.6}$$

In these fractions, the subscript *analysis* denotes the corresponding value of the network that is analyzed, and *random* denotes the respective value of the generated random network. The analyzed network is assumed to be small-world if $S > 1$ holds [HG08]. In this case, the clustering coefficient of the network that is analyzed is apparently higher than the one of the random network, and the respective average path length is smaller than the respective average path length of the random network.

Let us have a look at the example networks in Figure 2.6 again: For network A, $S = 0.77 < 1$ holds. Consequently, network A is not a small-world network. In contrast, for network B, $S = 2.98 > 1$ holds. So, network B has the small-world property. Due to the high clustering coefficient and the small average path length, every vertex can be reached from every other vertex via a short path, which is why this kind of network is called "small world". Singh [Sin10] suggested that small-worldness would be beneficial for developer networks since the small average path length and the high clustering coefficient allow a fast and efficient information flow between the developers that eases coordination among the developers.

Scale-Freeness *Scale-freeness* is a network metric that considers the degree distribution of the vertices in a network. The degree distribution of a network models the probability that a vertex has a specific degree [JNB03]. If a network is scale-free, its degree distribution obeys a power law [JNB03]. That is, in a scale-free network, there usually are many vertices that have a low degree, but few vertices that have a very high degree. This phenomenon is related to the so-called "heavy tail" of the power-law distribution: While for a random network, whose degree distribution is exponential, it is "extremely unlikely" that there is a vertex with an exceptionally high degree, it is likely in networks whose degree distribution obeys a power law [Job17]. In network theory, degree distributions are also used to build network growth models, that is, to build models that describe the evolution of a network over time and that model the probability for a vertex of a specific degree that it receives a new edge during the next evolution step [JNB03]. Barabási and Albert [BA99] found out that scale-free networks follow the principle of *preferential attachment*. That is, if a new vertex is added to a scale-free network, it will probably be connected to vertices that are already highly connected (i.e., that have a high vertex degree). In other words, scale-freeness means that there may be some central hub vertices due to the preferential-attachment principle. Suchlike hub vertices may play an important role within the network, and, thus, also in the system (i.e., OSS project) that is modeled by the network. The networks A and B in Figure 2.6 are scale-free, since their degree distributions follow a power law. Indeed, they have few vertices with very high degree, and many vertices with a rather low degree. However, in network C in Figure 2.8, we show a network that is not scale-free: All vertices have a similar, low degree, but no vertex has a high degree.

To determine whether a network is scale-free or not, one needs to check whether its degree distribution follows a power law. Usually, this is done by power-law fitting and a Kolmogorov-



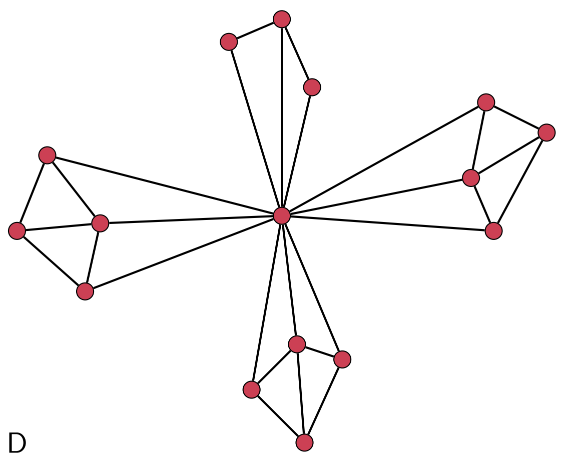


Figure 2.8: Networks C and D both have 16 vertices. Network C is *not* scale-free since its degree distribution does not follow a power law: All vertices have a similar degree, and there is no vertex that has a very high degree. In contrast, network D is scale-free. In addition, network D is hierarchical: There are only few vertices that have a very high degree (in particular, there is only one such vertex: the central vertex) but a low local clustering coefficient; and there are many vertices that have a low degree but a high local clustering coefficient (all the outer vertices that are tightly connected with each other). Furthermore, the central vertex in network D is the only vertex that connects the clusters, which are down in the hierarchy, with the central vertex, which is top in the hierarchy. This shows that there is a layered, hierarchical structure in network D.

Smirnov test. If the p-value of the Kolmogorov-Smirnov test is ≥ 0.05, then it cannot be rejected that the degree distribution of the network "is drawn from a power-law distribution" [CSN09] and, thus, the network is scale-free [DC13].

According to Dorogovtsev and Mendes [DM03], scale-free networks are robust to changes and, especially, if a random vertex is removed, the network stays connected without being split into separate sub-networks. However, this may not necessarily hold if a very central vertex is removed [CEA+01; JAM17]. For developer networks of OSS projects, scale-freeness is achievable, as there are steadily developers leaving or joining, which should not distort the information flow or coordination principles but should allow a successful continuation of the coordination and information flow in the project. Joblin et al. [JAM17] have shown that, for OSS projects that have considerably more than 50 developers, the corresponding developer networks usually are scale-free (after some initial turbulences at project initiation have disappeared).

Hierarchy Ravasz and Barabási [RB03] dealt with layered structures and detected a relationship between local clustering coefficient and vertex degree. *Hierarchical networks* are characterized by few vertices having high degree, but small local clustering coefficient, and a huge number of locally highly clustered vertices with small degree. More generally speaking, hierarchy describes how local groups of vertices are placed to each other [JAM17]. Similar to scale-freeness, the high-degree vertices also play a central role here, as they are in the upper part of the hierarchy. In network D in Figure 2.8, we show an example of an ideal form of a hierarchical network. Note that the concept of hierarchy in networks differs from a strict hierarchy in which there are only connections between different layers but not within layers [RB03]. That is, in a hierarchical network, there usually are edges between vertices on the same layer of the hierarchical structure, which lead to the high local clustering coefficient of the small-degree vertices that is inevitable for a hierarchical network by definition.

Hinds and McGrath [HM06] have shown that hierarchical organizational structures are beneficial for the coordination among geographically distributed people. For developer networks, Joblin et al. [JAM17] expected that hierarchy "indicates a centralized governance structure where decisions are primarily made at the top and passed down through a chain of command." That is, if developer networks exhibit hierarchical structures, this could be an indicator for the presence of organizational mechanisms to coordinate source-code changes among different groups of developers [JAM17].

In this thesis, we dedicate an entire chapter to hierarchical structures, in which we also provide additional background information on higher-order network structures (see Chapter 5).

Vertex Centrality In contrast to the above described network metrics that describe the structure and connectedness of a network in its entirety, we also consider network characteristics that characterize individual vertices (i.e., developers): An important characteristic for a vertex (and, in our setting, for a developer) is its centrality in the network, which captures the embeddedness and the importance of a vertex in the network [GNL16; NG04].

In the literature, many different network centrality metrics have been defined (see Grando et al. [GNL16]). From the large corpus of network centrality metrics, we selected three metrics that are established in the literature and which cover different aspects of centrality:

Degree centrality is a local centrality metric. It is equal to the total degree of a developer in the network, that is, the number of edges a developer has to other developers. The more connections a developer has, the higher their degree centrality [BE05; BEJ18].

Eigenvector centrality is a global centrality metric. It incorporates the centralities of the developers connected to a developer, to weight the importance of developers by the importance of others they are interacting with. Hence, a developer can have a high eigenvector centrality by either being connected to many other developers or by being connected to developers who also have a high eigenvector centrality [BE05; BEJ18; Bon07; Bon72]. For a vertex i, its eigenvector centrality x_i can be computed as $x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j$ where $N(i)$ is the set of neighbors of i (i.e., the vertices that are connected to i via an edge) and λ is a constant [BEJ18; Job17]. Using a vector $\mathbf{x}$ that stores all the centrality values x_i for all vertices in the network, we can rewrite this equation as $\mathbf{x} = \frac{1}{\lambda}\mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is the adjacency matrix of the network. The equation can then be transformed to the "eigenvector equation" $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ [Job17]. For details on how to solve this equation, we refer to Bonacich [Bon87].

Hierarchy centrality considers the community structure of a network. It is calculated by dividing the degree of a vertex by the local clustering coefficient of the vertex [RB03]. A high value in hierarchy centrality represents a developer having many connections to other developers, which in turn are loosely connected amongst each other. A low value stands for a developer down in the hierarchy, having only few connections to other developers but which are tightly connected among each other. Core developers should have a high hierarchy centrality due to their coordinative role [JAH+17; JAM17; JMA+15].

Hence, using network centrality metrics, we are able to assess the role and importance of a developer in the network, which represents the role of the developer in the organizational structure of the project.

In summary, all the described network characteristics, among others, help analyze developer networks with respect to the organizational structure of a project and the roles individual developers play in the project. Using these characteristics also eases the investigation of how developer networks change over time, which is also part of our research objectives.

2.4 Data Extraction and Data Processing

To perform our analyses on the coordination of OSS developers, we need to extract collaboration and communication data from the version-control system and the communication channel of interest. In the literature, many different tools have been proposed to extract collaboration data from source-code repositories [e.g., BKZ10; DP03; GSS21; SAB18]. For our purposes, we rely on a couple of tools that combine the extraction of collaboration data and communication data and also easily allow network constructions from the collected data. In particular, we rely on the tool Codeface[21] and its companion tool codeface-extraction[22].

Commit Data Codeface is able to extract commit metadata from a VCS, such as the author name and author date of a commit, as well as the committer name and committer date.[23] In addition, the extracted commit metadata also contain information about which files are changed in the commit, how many lines of code have been added or deleted, and what the corresponding commit message is, etc.

E-Mail Data Beside gathering commit metadata, Codeface is also able to extract e-mail header data from *mbox*[24] files, namely information about the author, the e-mail subject, and the date at which the e-mail was sent. To obtain the mbox files from the developer mailing lists that we want to analyze via Codeface, we use the tool nntp2mbox[25], which downloads the mbox files from the public mailing-list archive Gmane[26]. We then provide the mbox files as input to Codeface. In addition to the above mentioned header information, Codeface identifies e-mail threads by cross-referencing e-mail headers. That is, it considers the `<In-Reply-To>` and `<References>` tags in e-mail headers to group e-mails belonging to the same thread.[27]

21 `https://github.com/se-sic/codeface/` (accessed at 2023-06-28).
Codeface has already been used in previous work [e.g., HSA20; JA22; JAH+17; JAM17; JMA+15; MJT+22].

22 `https://github.com/se-sic/codeface-extraction/` (accessed at 2023-06-28)

23 The author date of a commit represents the point in time at which the author has created the commit. However, the committer date represents the point in time at which the committer has integrated the commit into the main repository. Whereas author and committer often differ when patches are sent to the mailing list and integrated into the repository by another developer later, they mostly are equal when commits are integrated via GitHub pull requests, for instance. In most of our analyses we only consider the author and the corresponding author date, as it represents the point in time at which the commit was initially created by the author.

24 The mbox file format is a file format that allows to store multiple e-mails (including their header information and attachments, etc.) in a single text file. Mailing-list archives such as Gmane allow to download a complete mailing list in mbox format. The mbox format is defined by the standard RFC 4155:
`https://datatracker.ietf.org/doc/html/rfc4155/` (accessed at 2023-06-28)

25 `https://github.com/xai/nntp2mbox/` (accessed at 2023-06-28)

26 `https://gmane.io/` (accessed at 2023-06-23). To search for archived mailing lists and find out which mailing lists are available on Gmane, their admin interface can be used: `https://admin.gmane.io/` (accessed at 2023-06-28)

27 In particular, Codeface uses the threading algorithm of the R package tm-plugin-mail in a fixed version taken from `https://github.com/bockthom/tm-plugin-mail/` (accessed at 2021-05-22), which basically implements the standard RFC 5256: `https://datatracker.ietf.org/doc/html/rfc5256/` (accessed at 2017-03-27)

Table 2.1: Extracted metadata from the different interaction channels.

Commit Data	E-mail Data	Issue Data
– Commit hash	– Message id	– Issue id
– Commit message	– E-mail subject	– Issue title
– Author date	– Author date	– Event date
– Author name	– Author name	– Author name
– Author e-mail address	– Author e-mail address	– Author e-mail address
– Committer date	– E-mail thread	– Event type
– Committer name	– …	– Additional information depending on event type
– Committer e-mail address		– Whether it is a PR
– Changed files		– Issue state (open/closed)
– Number of changed lines		– …
– …		

The commit data are extracted from the VCS, the e-mail data are extracted from mailing lists, and the issue data are extracted from GitHub. We only provide a selection of the extracted information that is relevant for this thesis.

Issue Data Codeface is also capable of extracting data from Jira issue trackers; yet, we do not extract Jira issues in this thesis, as already explained in Section 2.1.3.

For extracting issue data, we developed the tool GitHubWrapper[28], which queries metadata from GitHub's REST API[29]. Our tool runs separate queries for a single project to crawl data for issue events, reviews, review comments, and remaining comments from pull requests, as GitHub's REST API provides them via different interfaces of their API. Afterwards, GitHub-Wrapper combines the data from the different interfaces and performs additional postprocessing steps to unify the data (e.g., correctly determine the actual actor of a "subscribed" event, as on GitHub such an event can either be actively triggered by a user subscribing themselves to an issue or, passively, by being mentioned by someone else) or to gather additional information (e.g., extracting names and e-mail addresses from referenced commits). The extracted issue metadata contain information on which GitHub user has triggered which event (commented, labeled, merged, etc.), on which issue or pull request, and at which point in time. Note that we remove all events that have been triggered by the "ghost" user, because it is an replacement for all GitHub users that have been deleted.[30] Keeping the "ghost" user would distort our analyses, as its activities correspond to all deleted users together and not to a single user.[31]

In Table 2.1, we provide an overview of the different information that we extract from the VCS (commit data), from the mailing lists (e-mail data), and from GitHub issues (issue data) via the aforementioned tools.

Bot Detection When analyzing GitHub issue data, one also needs to take care of automated, non-human actors, since many GitHub projects use automatic bots, which, for instance, submit comments to issues or review pull requests. Bots often also close abandoned issues, execute automatic code refactorings, run continuous-integration tasks, or perform similar tasks [BP19; GLD+20; MUD+19; WB19; WS20; WSS+18]. Previous research has shown that about 20% of

28 `https://github.com/se-sic/GitHubWrapper/` (accessed at 2023–06-28)
29 `https://docs.github.com/en/rest/` (accessed at 2022-03-14)
30 `https://github.com/ghost/` (accessed at 2024-01-22)
31 `https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-your-personal-account/deleting-your-personal-account/` (accessed at 2024-01-22)

issue comments are posted by bots and about 31% of pull requests involve bots [GDL+21]. Not to distort developer-network characteristics and the analysis of issue data with respect to developer's collaboration and communication activities, in general, we must detect and remove bots' activities from the extracted issue data. Whereas some bots are labeled by GitHub to be bots or use a "bot" suffix in their usernames, research has found that many bots are not labeled as such and also human users can use the "bot" suffix, which makes bot detection a non-trivial task [GDL+21]. For that reason, Golzadeh et al. [GDL+21] have developed the tool BoDeGHa[32] to automatically detect bots based on the total number of comments and commenting patterns, as bots' comments are created automatically and follow certain patterns. Unfortunately, many projects use predefined issue or pull-request comment templates[33], making BoDeGHa misclassify human users as bots when they mostly use templates or post stereotyped comments. As a consequence, after using BoDeGHa for automatic bot detection on GitHub users that have posted, at least, 2 000 comments in a project, we perform some manual adjustments on a sample basis. For example, we classify 83 users that are widely-used bots as a bot even if the automatic bot detection did not classify the user to be a bot (e.g., the "lockbot"[34], which is a bot that automatically locks issues but usually does not post comments and, therefore, is not detectable by BoDeGHa). In particular, we manually checked all usernames that had a "bot" substring (such as the "lockbot") whether they are marked as a bot by GitHub. If so, we added them to the set of bots identified by BoDeGHa.

Combining Data As Codeface stores the commit data and e-mail data in a database, whereas GitHubWrapper and BoDeGHa have their own output files, there is a need to combine the data from the different sources and unify the different data formats via some post-processing. To do so, we use the tool codeface-extraction, which ends up in dumping unified comma-separated-value (CSV) files for author, commit, e-mail, issue, and bot data. codeface-extraction is also able to dump all these data in a pseudonymized form that we use to make the collected data available in replication packages for the individual studies that we perform.[35]

Developer Disambiguation To match developers, Codeface and codeface-extraction use a common id service to match developers who use the same name or the same e-mail address.[36] This way, we can match developers that use another name on GitHub than they use to create commits in the VCS, but use the same e-mail address in both interaction channels, for example. Developers found in only one of the data sources are kept, though. To match developers via names and e-mail addresses, Codeface and codeface-extraction use the disambiguation heuristic of Oliva et al. [OSO+12], which has been proved to be accurate [WSS+16]. On top of this, we perform additional sanity checks and manual corrections

32 `https://github.com/mehdigolzadeh/BoDeGHa/` (accessed at 2023-06-28)

33 There are various different means of how issue templates can be used and enforced by a project, which also varies during project evolution. One example can be found in the official documentation of GitHub: `https://docs.github.com/en/github/building-a-strong-community/about-issue-and-pull-request-templates/` (accessed at 2022-03-14)

34 `https://github.com/apps/lock/` (accessed at 2023-06-28)

35 For our studies that we describe in the following chapters, we provide the pseudonymized raw data on the corresponding supplementary websites. We collect all the links to the supplementary websites in Appendix A.

36 As some GitHub users keep their real names or e-mail addresses private, we only use their public usernames if no name or e-mail is available on GitHub.

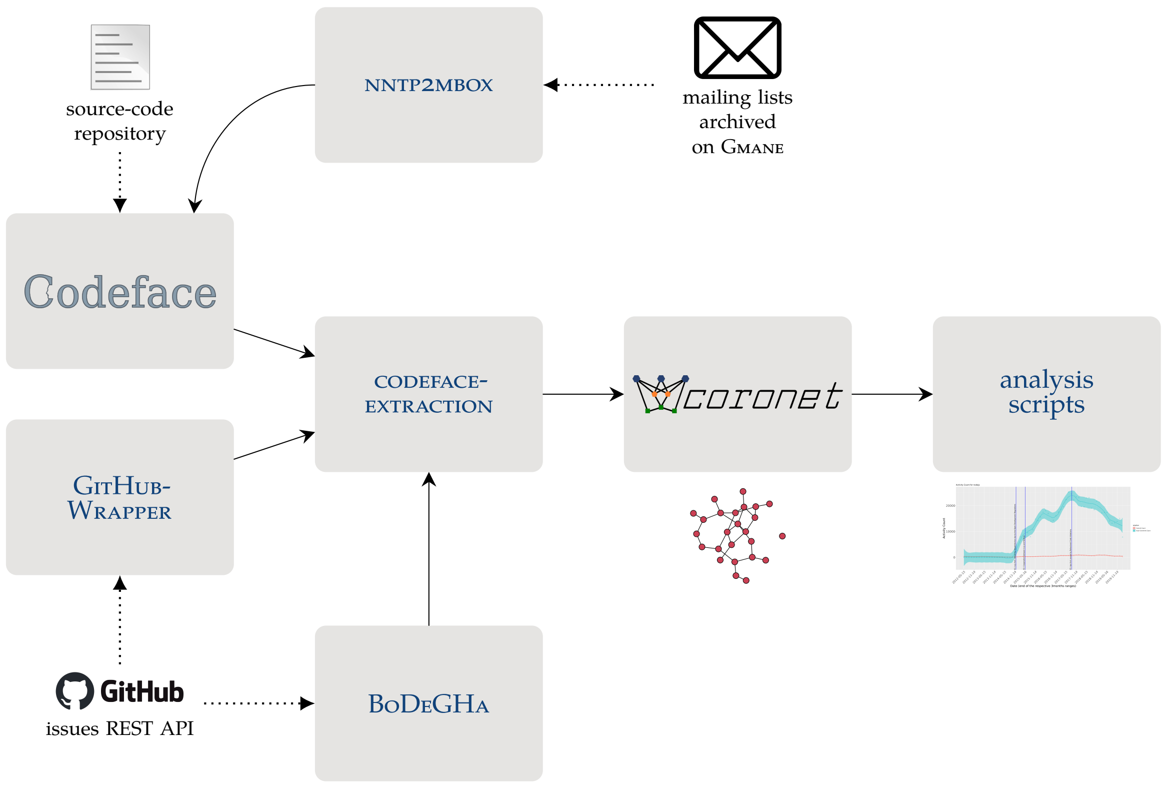


Figure 2.9: Overview of our data-extraction and data-processing toolchain: CODEFACE processes mailing lists (which we have downloaded from publicly accessible archives via NNTP2MBOX) and the source-code repository, whereas our GITHUBWRAPPER and the bot-detection tool BoDeGHa [GDL+21] extract data from the GitHub issues REST API. Then, CODEFACE-EXTRACTION combines and postprocesses all the data. Afterwards, CORONET processes all the data and constructs networks. Finally, we use the data processed by CORONET in our specific analysis scripts.

in cases where the automatic disambiguation heuristic is not able to match developers (e.g., due to special characters in their names or misspelled names).

Developer Network Analysis To perform our actual analyses on developer coordination and organizational structures in OSS projects, we use the network library CORONET[37], which basically reads the CSV files that were generated by CODEFACE-EXTRACTION beforehand, processes all the data, and provides functionality to build various kinds of developer networks (as already described in Section 2.3). Within CORONET, we also remove all bot-triggered events from the extracted GitHub issue data.[38] Finally, we use different analysis scripts that use the functionality offered by CORONET for the specific analyses that we perform in this thesis.

In Figure 2.9, we provide an overview of our entire data-extraction and data-processing toolchain for the different sources of information (i.e., the different interaction channels) that we analyze throughout this thesis.

37 `https://github.com/se-sic/coronet/` (accessed at 2023-06-28)
38 Note that bot removal is not part of the default configuration of CORONET (at least, not up to its version 4.4) and needs to be specifically enabled in CORONET's project configuration.

3 Synchronous Development in Open-Source Software Projects

This chapter shares material with Bock et al. [BHJ+22].

The success of large software projects relies on the extent to which developers coordinate their efforts. This is especially true for large-scale open-source software (OSS) projects, to which often numerous globally distributed and independent developers contribute [Her07]. When multiple developers contribute to interrelated source-code fragments, changes that lack coordination often introduce unintentional side effects. Developers must coordinate their interdependent activities to prevent conflicting changes, to avoid bugs, or to keep the source code simple and maintainable [Bir11; CHC08; CMR+09; KSD11]. In large-scale projects, developer coordination is absolutely crucial to ensuring high-quality software and to supporting high developer productivity [CH13].

Since software developers in OSS projects are often globally distributed, they mostly communicate via the Internet to discuss software issues or enhancements or to review source-code changes [WGS03]. Mailing lists, issue trackers, and instant messengers are the most commonly used communication channels for coordination of developers in OSS projects [SSF+17]. In this study, we dedicate attention to analyzing developer communication on mailing lists because they are historically rich and well-established sources of data for discussions regarding software architecture and reviewing of source-code changes [RGS08; RLM19]. In a recent study on 37 OSS projects, Mannan et al. [MAJ+20] have shown that about 89% of such discussions take place on the project's mailing list. Mailing lists are a greater source of longitudinal data than more recently introduced social coding platforms (e.g., GitHub), because their usage dates back more than 10 years (see Table 3.2). Mailing lists are also used to discuss the outcomes of developer conferences and similar events where complex issues and long-term plans for feature development are discussed. Even developers in OSS projects who work for corporations may use mailing-list discussions to communicate their intentions to others as public communication is one of the basic concepts in OSS projects [Rie15].

To obtain deeper insights into the fundamentals of developer coordination and the role communication plays in OSS projects, we investigate the relationship between co-editing activities on source-code artifacts and communication activities on the developer mailing list. For this purpose, we replicate and extend an empirical study of Xuan and Filkov [XF14] on synchronous development in OSS projects, which we will refer to as the *original study*. The authors of the original study identified pairs of developers co-editing files to explore the relationship between developer productivity and communication activities. Their major finding was that time intervals rich in co-editing activities are correlated with time intervals rich in e-mail activities and, more importantly, that during these synchronized periods developer productivity was higher.

The original study already provided interesting and useful insights on developer collaboration and developer communication. Nonetheless, they relied on a rather low-level, simplistic view on communication, as all e-mails sent to the mailing list within a specific time period are considered equally likely to be related to each other. Compared to the original study, we take a more nuanced view on communication activity by grouping individual e-mails together according to the thread of communication they belong to. That is, we extend the original study by lifting the message-based view of developer communication to a *conversation-based* view, which incorporates the context of e-mails by grouping e-mails according to threads. Since e-mails belonging to the same thread address a relatively narrow topic space, the likelihood of these e-mails being content-wise related is higher [BPD+08]. A heuristic solely based on temporally close-by e-mails sent to the mailing list likely misses meaningful communicative associations between developers. Hence, we investigate the question of whether there is a difference in the dependence of social and technical activities using a message-based or a conversation-based view of the complex processes involved in developer coordination.

By means of an empirical study, we investigate whether the different abstraction levels of developer communication (message-based vs. conversation-based) affect the relationship between commit activity and e-mail communication observed in the original study using state-of-the-art time-series analysis. Knowing about differences between abstraction levels could be exploited for improving developer coordination (e.g., to predict on which parts of the source code a developer is likely to work on next). In particular, we investigate whether synchronous development is temporally aligned with coordination on the mailing list. To find out whether developers working on the same file contemporaneously also communicate, that is, to measure synchronization, we use dynamic time warping [RJ93], a state-of-the-art time-series analysis technique.

It is important to note that, when we investigate whether co-editing activity is accompanied by communication on the mailing list, we cannot be sure that the mailing-list communication is related to the co-editing activity. However, it is a difficult task to find out which e-mails are related to the co-editing activity and which not, as e-mails sent by a developer shortly before or after a commit could also cover completely unrelated topics (especially if there are many commits and e-mails within a short period of time); when relating only e-mails whose subject is related to the commit, we may omit related e-mails that have a different subject. For that reason, we propose two different approaches, which we call the lower-bound approach and the upper-bound approach: Whereas the *upper-bound approach* considers all e-mails sent to the mailing list to identify time intervals rich in e-mail activities (as in the original study), the *lower-bound approach* considers only e-mails whose subject is topically related to the co-editing activity following a very strict matching procedure. We call them upper-bound and lower-bound because the former considers all messages without restrictions, ending up in the maximum amount of considering communication activity, and the latter considers only messages related to co-editing activity, which is a very small subset of the total set of e-mails. Hence, the actual amount of the communication that is content-wise related to the co-editing activity lies in-between these bounds. For the upper-bound approach, we additionally perform manual checks to explore to which extent the content of e-mail communication is related to temporally close-by collaboration on the source code.

For the purpose of the study, we analyze a combined history of 40 years of data for three highly active and widely deployed OSS projects: QEMU, BusyBox, and OpenSSL. We investigate synchronous collaboration on source code and coordination on mailing lists using different abstraction levels. Overall, we found evidence that a more abstract and higher-level view of developer communication describes developer coordination more accurately than a less abstract and more simplistic view.

In summary, we make the following contributions:

- A replication of the original study on a different dataset: three highly active and widely deployed OSS projects. Regarding the existence of synchronous development, we are able to confirm the results of the original study. However, we cannot confirm the results of the original study regarding code growth and implementation effort in synchronous development nor the relationship between the number of synchronous collaboration activities and the number of synchronous communication activities.
- A method for raising the abstraction level of exploring synchronization between developers' collaboration and communication activities: We lift the view of developer communication from a message-based model, which treats each e-mail individually, to a conversation-based model, which is semantically richer due to grouping e-mails that represent conceptually related discussions.
- A continuous variable *synchronicity degree* to quantify the significance of co-editing artifacts. (Previously only binary variables were used.)
- An upper bound and a lower bound for determining whether e-mail communication is related to co-editing activity, as relating e-mail communication to co-editing activity is not trivial.
- A manual investigation whether e-mail communication is content-wise related to temporally close-by collaboration activities. The results of our qualitative analysis indicate that only between 29% and 47% (depending on the subject project) of temporally aligned collaboration and communication activities are content-wise related.
- A novel technique based on dynamic time warping (DTW) to measure synchronization of activities across source code and mailing lists to adequately take care of the dynamic nature of socio-technical congruence.
- An extensive empirical study of three highly active and widely deployed OSS projects. Our results indicate that a more abstract and higher-level view on communication leads to a stronger statistical dependence between developers' pairwise technical activities than a less abstract, simplistic view.
- A replication package including pseudonymized raw data, analysis scripts, and results on a supplementary website:[1]
  `https://se-sic.github.io/paper-coordination-bursts/` and
  `https://zenodo.org/record/5131282`.

1 Note that the way in which the replication package is presented on the supplementary website slightly deviates from what we present in the following sections, since the website belongs to Bock et al. [BHJ+22], which also covers additional research questions that are out of scope of this thesis. Nevertheless, beside presenting additional data and analyses that are not present in this chapter, the results that we present on the supplementary website and the ones we present in this thesis are identical.

3.1 Background & Related Work

Xuan and Filkov [XF14] defined *synchronous development* as the situation where two developers contribute to the same source-code file within a short period of time. In the original study, they considered two different kinds of synchronous activities: co-commit bursts and e-mail bursts. To explore the temporal relationship between co-commit bursts and e-mail bursts, they constructed continuous curves by smoothing time series of bursts. In the end, they calculated the correlation between these curves to measure the synchronization of collaboration activities and communication activities.

In this section, we first introduce the algorithms and concepts of co-commit bursts and e-mail bursts as well as the corresponding continuous curves in detail, as used by the authors of the original study. Afterwards, we provide an overview of related work in which the relationship between source-code changes and communication activities has been investigated. Thereafter, we briefly introduce the concept of dynamic time warping, which is an established time-series analysis technique, which we apply in our study to investigate the temporal correlation between co-commit bursts and e-mail bursts.

3.1.1 C-Bursts, E-Bursts, and the Corresponding Curves

Co-Commit Bursts (C-Bursts) When two developers commit to the same source-code artifact (i.e., file) within a short period of time, Xuan and Filkov [XF14] called this a co-commit burst (short, *C-burst*). For two commits to be included in a burst, the time difference between the commits must not exceed a specified *time window*, denoted by ξ. The time window resembles the fact that developers may have different preferences of how quickly and how often they contribute code. Note that looking at only pairs of developers is not a limitation, as groups of more than two collaborating developers end up in separate C-bursts for each pair of developers that are part of such a group. Hence, group-wise collaboration can be considered as the composition of the collaborations of individual developer pairs.

As we describe in Algorithm 3.1 (adapted from Xuan and Filkov [XF14]), for each pair of developers (Lines 2–22), it is checked whether the two developers are authors of mutual commits to the same source-code artifact that have a time[2] distance of, at most, ξ, and whether these commits have been made to, at least, one common artifact (Line 7). If so, these commits form a C-burst (Lines 4–10), where each burst is represented by a start time and an end time. Finally, overlapping bursts of the same developer pair are merged (Lines 11–19). This algorithm has a complexity of $\mathcal{O}(|\overline{D}|^2 \cdot |\overline{c_{max}}|^2)$, with $|\overline{D}|$ being the number of developers and $|\overline{c_{max}}|$ being the maximum number of commits of a single developer in the project.

In Figure 3.1, we show an example of four commits made by one pair of developers, $D1$ and $D2$. In the commits c_1 and c_2, both $D1$ and $D2$ change artifact $A3$. Using a time window $\xi = 5$ days, c_1 and c_2 were created within the time window and form a C-burst. Analogously, c_2 and c_3 form a C-burst due to the change of artifact $A5$. Since both bursts overlap at c_2, they are merged into one burst. c_4 also changes the same artifact as c_3, but these commits have a time distance that is larger than the time window. Hence, c_3 and c_4 do not form a C-burst.

2 All timestamps are transferred to the Coordinated Universal Time (UTC) first.

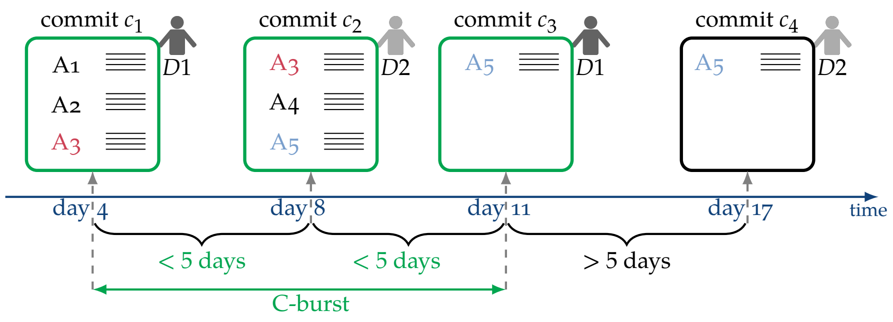


Figure 3.1: An example containing four commits made by one pair of developers, $D1$ and $D2$. Commits c_1, c_2, and c_3 form a C-burst: In c_1 and c_2, both developers change one artifact synchronously within the time window ξ of 5 days; commits c_2 and c_3 also form a burst for the same reason. Since these two bursts overlap at c_2, they are combined to one burst. c_3 and c_4 do not form a C-burst as their temporal distance is larger than the time window.

Algorithm 3.1 Identification of C-bursts

```
Input: list of commits $\bar{c}$ (annotated with timestamps and developer names)
1: $bursts \leftarrow \emptyset$
2: for each pair of developers $\{A,B\}$ do
3:   $bursts_{AB} \leftarrow \emptyset$
4:   for each commit $c_A \in \bar{c}$ authored by developer $A$ do
5:     $burst \leftarrow \{c_A\}$
6:     for each commit $c_B \in \bar{c}$ authored by developer $B$ do
7:       if $|\text{time}(c_A) - \text{time}(c_B)| \leq \xi$
         and $\text{artifacts}(c_A) \cap \text{artifacts}(c_B) \neq \emptyset$ then
8:         $burst \leftarrow burst \cup \{c_B\}$
9:       end if
10:    end for
11:    if $burst \neq \{c_A\}$ then
12:      for each burst $b \in bursts_{AB}$ do
13:        if $\text{overlap}(burst, b)$ then
14:          $burst \leftarrow \text{merge}(burst, b)$
15:          $bursts_{AB} \leftarrow bursts_{AB} \setminus \{b\}$
16:        end if
17:      end for
18:      $bursts_{AB} \leftarrow bursts_{AB} \cup burst$
19:    end if
20:  end for
21:  $bursts \leftarrow bursts \cup \{bursts_{AB}\}$
22: end for
Output: $bursts$ containing lists of C-bursts for all developer pairs
```

In addition to identifying C-bursts, the original study analyzed how C-bursts are related to code growth ΔL and implementation effort ΔW, defined as follows: Let L_{Add} denote the number of added lines of code (LOC) per commit and L_{Delete} the number of deleted LOC per commit. Then, $\Delta L = L_{Add} - L_{Delete}$ and $\Delta W = L_{Add} + L_{Delete}$ [XF14].

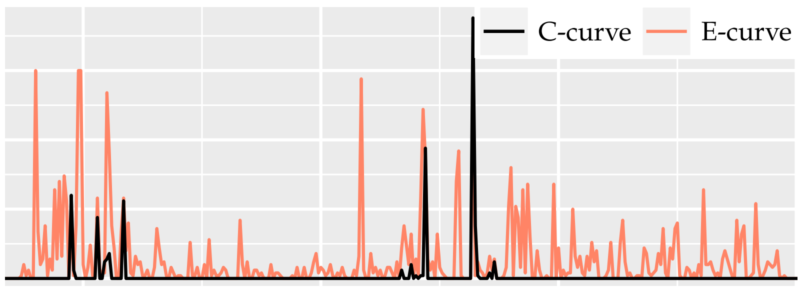


Figure 3.2: Example of the C-curve and the E-curve of a pair of developers. The horizontal axis represents the time dimension (days), the vertical axis the intensity of the bursts (number of commits and e-mails, respectively, within the burst).

E-Mail Bursts (E-Bursts) Xuan and Filkov [XF14] used a message-based model to identify e-mail bursts. An e-mail burst (short, *E-burst*) arises if two persons each send an e-mail to the mailing list within a defined time window ξ. For determining E-bursts, Xuan and Filkov used almost the same approach as for identifying C-bursts: For each pair of developers, iterate over all the e-mails sent by one developer and search for all e-mails of the other developer whose creation dates have an absolute time difference of less than or equal ξ to the e-mail of the first developer. As opposed to the C-burst identification, there are no further conditions to be checked. Hence, all detected e-mails of two different developers within the time window ξ form an E-burst, where each burst is represented by a start time and an end time. Similar to C-bursts, overlapping E-bursts of the same developer pair are merged.

C-Curves and E-Curves To check whether two developers coordinate their collaboration, that is, to check whether C-bursts and E-bursts of a developer pair are synchronized, Xuan and Filkov [XF14] introduced the notions of C-curves and E-curves. They computed a C-curve (or E-curve, respectively) for each developer pair denoting the number of commits (or e-mails, respectively) that are part of a burst, aggregated for each day of the time series, as we illustrate in Figure 3.2. By comparing the C-curve and the E-curve of a developer pair, they investigated whether synchronous development and communication activities of the developer pair are temporally related. Since coding collaboration and e-mail communication do not take place at exactly the same time, it is not useful to directly compute the overlap of the resulting curves. Therefore, they applied Gaussian smoothing on each of the curves to also be able to align slightly off-set C-bursts and E-bursts. To compare the smoothed curves, they used the Pearson correlation coefficient to check whether the C-curve and the E-curve of a developer pair are dependent or independent of each other.

3.1.2 Investigating the Relationship Between Coding and Communication

Beside the study of Xuan and Filkov [XF14], which we reproduce and extend, there has been various research on the relationship between coding and communication. Herbsleb and Grinter [HG99] conducted a study on coordination in geographically distributed software projects. They found that communication between developers is one of the most important parts of today's well-working software development. Related studies [CH05; CH13; HM03b; MFH02; SFD05] showed that coordination in software projects affects software quality and that considering social aspects, such as communication of developers, is essential for understanding OSS projects. We extend on these by considering the temporal and content-wise dependency between communication and technical activities from a higher-level perspective.

The authors of the original study enhanced their work by identifying "work-talk patterns" [XDF16] on the time series of working activities (i.e., commits to the source code) and communication activities (i.e., response e-mails on the mailing list) for developer pairs. Their identified patterns indicate that collaboration on source-code artifacts and coordinating events on the mailing list are temporally related [XDF16].

Gharehyazie and Filkov [GF17] extended the original study by not investigating pairs of developers but groups of developers working on the same source-code artifacts temporally close-by. In their work, they analyzed whether the size of such groups is purely random and how often developers work in groups rather than working alone. Contrary to our work, they chose an even more coarse-grained level of abstraction and analyzed source-code changes on package level. Similarly to our qualitative analysis, they manually checked for a content-wise relationship between e-mails and source-code changes of a developer group. To do so, they searched for file names of temporally close-by edited files within e-mails of the developer group (which is also one of the three approaches that we use in our qualitative analysis). As a result, they found that there is actual coordination of source-code changes on the mailing list. In addition, they performed developer surveys which confirmed their results, which is also in line with the hypotheses of our study. They also analyzed if the code growth is higher and the corresponding effort is lower for developers working in groups than solely. In line with our results, they identified only rare projects where this hypothesis holds [GF17].

Researchers also used network approaches to describe the collaboration and coordination of developers: López-Fernández et al. [LRG+06] constructed networks representing mutual contributions of developers to the same software module (i.e., files contained in the same directory). Jermakovics et al. [JSS11] built networks based on co-editing files, and Toral et al. [TMB10] analyzed social communities on e-mail networks that arose from software development. Joblin et al. [JMA+15] constructed developer networks based on co-commits on source-code artifacts, especially at the more fine-grained level of functions, and used network analysis techniques to gain more information on collaboration. Joblin et al. [JAM17] also used a network approach to analyze evolutionary trends of developer coordination. However, synchronicity between collaboration at the code level and communication on the mailing list was not in their focus. Bacchelli et al. [BDL10] investigated the e-mail communication of OSS projects, determined the much discussed source-code artifacts, and investigated their defect-proneness. Bird et al. [BGD+06; BPD+08] analyzed e-mail social networks and examined whether coding activities on files within sub-communities are related to communication on the mailing list. In our study, we measure the synchronicity of co-edits and directly compare synchronous development on files to e-mail communication using state-of-the-art time-series analysis.

Jiang et al. [JAK+14] linked commits to specific e-mails on the mailing list by comparing commits to previously submitted patches on the mailing list. Also, Ramsauer et al. [RLM19] used a similar approach. Compared to our study, they explicitly traced back commits (i.e., changed source-code lines) to e-mails that contain these changed source-code lines as part of a patch, whereas we are interested in all kinds of e-mails that are in some way related to specific code changes, not necessarily containing patches that change exactly the lines that are changed in a commit. Coordination of specific changes may cover more discussions and high-level design decisions than just submitting or discussing patches. Ali et al. [AGA13] mined software repositories to build traceability links between source code and textual requirements documents using advanced information-retrieval techniques. As we look for relationships between code changes and communication among developers, we also link textual documents (sets of e-mails) to source-code changes, but as we are more interested in the coordination activities than the concrete content, we do not use advanced information-retrieval techniques.

3.1.3 Dynamic Time Warping and Sakoe-Chiba Band

To investigate the relationship between two time series (such as the C-curve and E-curve displayed in Figure 3.2), the technique of *dynamic time warping* (*DTW*) has been developed in the context of speech recognition (e.g., to quantify differences in pronunciation of spoken language), but has also been applied in other contexts, such as biometrics analysis or time-series analysis in medical settings [BC94; Gio09; KP01; RJ93]. In general, DTW tries to align two time series of equal length with each other by traversing a matrix D beginning in $D(0, 0)$ and ending in $D(n, n)$, where 0 and n are the earliest and latest time of the two time series. Using dynamic programming and calculating cumulative sums of distances on the path, DTW explores the whole matrix space to find the path of the shortest distance [BC94; KP01; RJ93] (for more details, see Rabiner and Juang [RJ93] or Giorgino [Gio09]). As a further restriction to DTW, the *Sakoe-Chiba band* only allows exploring cells in the matrix at which the absolute distance of the compared data is less than or equal to the chosen band-window size. So, only data points of the two time series get matched that have an absolute distance less than or equal to the chosen band-window size [SC78]. In our study, we use DTW to compare C-curves and E-curves, and the Sakoe-Chiba band prohibits that a C-burst and an E-burst that occur temporally extremely distant to each other get matched by the DTW algorithm. We provide more details on how we apply DTW and the Sakoe-Chiba band below in the description of our research approach.

3.2 Research Approach

In our study, we extend the original study by lifting the abstraction level of mailing-list communication and by changing the methodology of comparing C-curves and E-curves. Additionally, we introduce a metric to quantify the synchronicity of C-bursts.

On mailing lists, we differentiate between message-based communication (considering all synchronously sent e-mails from two developers) and conversation-based communication (considering only e-mails belonging to the same thread). When identifying E-bursts, we use two different approaches to determine a lower-bound and an upper-bound for identifiable coordination. Finally, we use a sophisticated time-series analysis technique to check whether C-bursts and E-bursts of a pair of developers are synchronized.

3.2.1 Research Questions

To obtain deeper insights into the fundamentals of developer coordination in OSS projects, we investigate the relationship between co-editing activities on source-code artifacts and communication activities on the mailing list. The idea is that developers rely on the characteristic information conveyed by conversation threads for building a mental model of the software and the processes around it, which in turn drives the communication and coordination with other developers [CSC93; EKL+01; SCE+08]. So, the overarching question is whether there is a difference in the statistical dependence of social and technical activities between a semantic, high-level view and a rather simple, low-level view of the complex processes involved in developer coordination. That is, we investigate whether a conversation-based representation of developer coordination reveals a statistically stronger relation to co-editing source-code artifacts than a sole message-based representation. Specifically, we will address the following research question regarding the abstraction level of coordination:

> **RQ:** Which abstraction level of the mailing list captures the coordination of developers best: message-based communication or conversation-based communication? That is, which of the two abstraction levels of the mailing list leads to identifying a stronger statistical dependence between technical activities and social activities on the mailing list?

To answer our research question, we make use of the concepts of C-bursts and E-bursts from the original study, since C-bursts depict synchronous technical activities and E-bursts depict synchronous social activities of a developer pair. Nevertheless, as we use another set of projects than the original study, we first need to demonstrate that their approach also works on our set of projects, before we can lift their approach to different abstraction levels of developer communication. Therefore, as in the original study [XF14], we investigate whether C-bursts or E-bursts are just artifacts of a purely random process (and, thus, uninteresting for us, as we expect that collaboration and communication are dependent and correlated processes). Following the research questions from the original study, we check whether empirically observed C-bursts occur more frequently than purely random. While the original study relied on only the existence of C-bursts, we also check whether the empirically observed C-bursts have a higher synchronicity degree than they would have by chance. As the original study examined whether synchronous development leads to a higher code growth and less implementation effort than in non-synchronous development, we also investigate these phenomena in our study. Similar to the original study, we also analyze whether the relationship between the number of C-bursts and the number of E-bursts differs from a purely random process. We provide more information on how we perform all these checks and how we address our new research question below in Section 3.3. Prior to this, we explain in the following which concepts and methods we use for answering our research question.

3.2.2 File-Based C-Bursts

To extract C-bursts for a developer pair, as defined in Section 3.1.1, we rely on the abstraction level of files as jointly edited source-code artifacts. That is, the commits from two developers within a specific time window form a C-burst if the commits change the same file. One could also think of considering a C-burst if the commits just change a file in the same folder, as files in the same folder may be semantically related to each other. However, projects differ in how they organize files into folders. Folders may be deeply nested, having files at different nesting levels.

Table 3.1: Examples of the synchronicity degree deg_{sync} for different numbers of added LOC by developers A and B in C-burst c.

$\text{add}(A, \text{syncArt}(c))$	$\text{add}(B, \text{syncArt}(c))$	$\text{add}(A, \text{art}(c))$	$\text{add}(B, \text{art}(c))$	$\text{deg}_{\text{sync}}(c)$
10	10	10	10	1.00
10	10	20	20	0.50
10	10	15	2 010	0.06

High-level folders may be too coarse-grained (co-editing source code in the same folder may be not related at all), whereas low-level folders may be too fine-grained (missing the relations between files at different levels of nested folders). As it is not obvious and mostly project-dependent which nesting level of folders would be appropriate for C-burst identification, we stick to a file-based analysis, which has been established in the original study.

3.2.3 Synchronicity Degree

The method to identify synchronous development described in Section 3.1.1 is limited because it does not quantify the magnitude of the overlap among the commits of a C-burst. Essentially, the variable denoting synchronous development is binary. To gain precision, we model the overlap of synchronously changed artifacts within a burst using a continuous variable. This is beneficial because synchronous commits from two developers can contain changes to one common artifact while most of the other changes are to artifacts that are touched by only one of the developers [BNM+11]. For this reason, we introduce the *synchronicity degree*, a metric capturing the overlap based on the number of LOC each of the two developers adds to the artifacts changed in a C-burst. We calculate the synchronicity degree individually for each C-burst. Formally, we define the synchronicity degree deg_{sync} for a C-burst c of the developers A and B as follows:

$$\text{deg}_{\text{sync}}(c) = \sqrt{\frac{\text{add}(A, \text{syncArt}(c))}{\text{add}(A, \text{art}(c))} \cdot \frac{\text{add}(B, \text{syncArt}(c))}{\text{add}(B, \text{art}(c))}}, \tag{3.1}$$

where $\text{add}(A, x)$ denotes the number of code lines added by developer A to the list of code artifacts x in C-burst c, $\text{syncArt}(c)$ denotes the list of synchronously changed artifacts in C-burst c (i.e., the set of all artifacts changed by both A and B in their respective commits), while $\text{art}(c)$ is the set of all artifacts changed in C-burst c. In other words, to determine the synchronicity degree, we calculate the geometric mean of the code changes made by the two developers involved in a C-burst. Specifically, the metric incorporates the size of changes to synchronously changed artifacts of each developer, normalized by the changes to all artifacts in the C-burst. To let the synchronicity degree assign high values only to C-bursts that have a high portion of synchronously changed artifacts, and to down-weight C-bursts that have a highly imbalanced number of changes to non-synchronously changed artifacts, we use the geometric mean, this way reducing the weight of higher values compared to the arithmetic mean (as we also show in the following examples).

In Table 3.1, we provide examples of how the synchronicity degree treats the size of mutual contributions in a C-burst: If both developers change all artifacts touched in the commits of

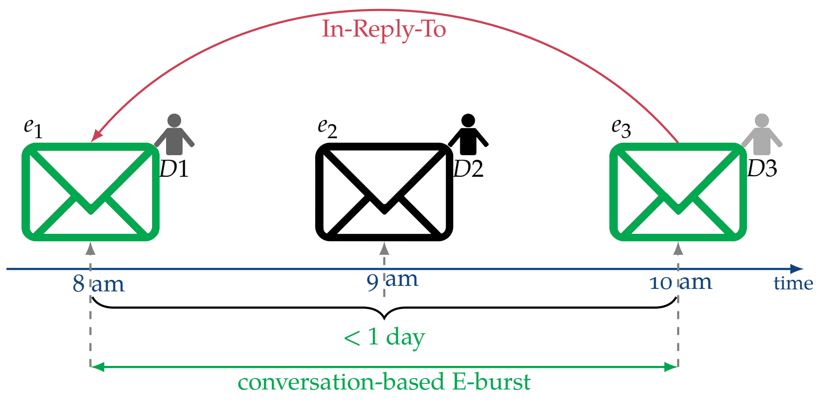


Figure 3.3: An example containing three e-mails to the mailing list sent by three different developers $D1$, $D2$, and $D3$. All three e-mails were sent on the same day; therefore, each pair of e-mails forms a message-based E-burst of the involved developer pair. Incorporating thread information, we see that e_1 and e_3 belong to the same thread since e_3 is sent in reply to e_1. Therefore, e_1 and e_3 form a conversation-based E-burst.

a C-burst synchronously, $deg_{sync} = 1$. When both developers change synchronous artifacts and individually changed artifacts of the C-burst in a balanced way, $deg_{sync} = 0.5$. Finally, if the proportion of synchronously added lines over all added lines is highly imbalanced, the synchronicity degree is very low (e.g., $deg_{sync} = 0.06$).[3]

3.2.4 Message-Based and Conversation-Based E-Bursts

We analyze the mailing list of the selected software projects by identifying message-based E-bursts, as described in Section 3.1.1. For identifying conversation-based E-bursts, we introduce the additional constraint that only e-mails belonging to a common thread can appear in an E-burst. The rationale is, if two e-mails belong to the same e-mail thread, then this is a more reliable indicator of coordination due to the topical scope in e-mail threads. We identify e-mail threads by cross-referencing e-mail headers, as described in Section 2.4.

For example, consider the situation illustrated in Figure 3.3: Developers $D1$, $D2$, and $D3$ each write an e-mail to the mailing list on the same day. Without considering thread information, each pair of e-mails forms a message-based E-burst of the corresponding developer pair, as all three e-mails were sent on the same day. However, e-mail e_2 may address a completely different topic than e_1 and e_3; e_2 may not be related to e_1 and e_3 at all. When considering thread information, we see that e_1 and e_3 belong to the same thread since e_3 is sent in reply to e_1. Therefore, these e-mails can be considered as content-wise related (as defined by the thread). As a consequence, we consider e_1 and e_3 as a conversation-based E-burst, which is a more robust indicator of coordination between developers $D1$ and $D3$ due to the conceptual relation of their e-mails. In this example, there is no conversation-based E-burst including e_2.

3 In this example, only about 0.5% of developer B's changed lines are made to a synchronously changed artifact, meaning that there is almost no synchronicity. If we would use the arithmetic mean instead of the geometric mean for the calculation of the synchronicity degree, we would get a value of 0.34 instead of 0.06, as the imbalance of the non-synchronously changed lines would not be respected. As a consequence, we use the geometric mean, since a value of 0.06 better describes that there is almost no synchronicity.

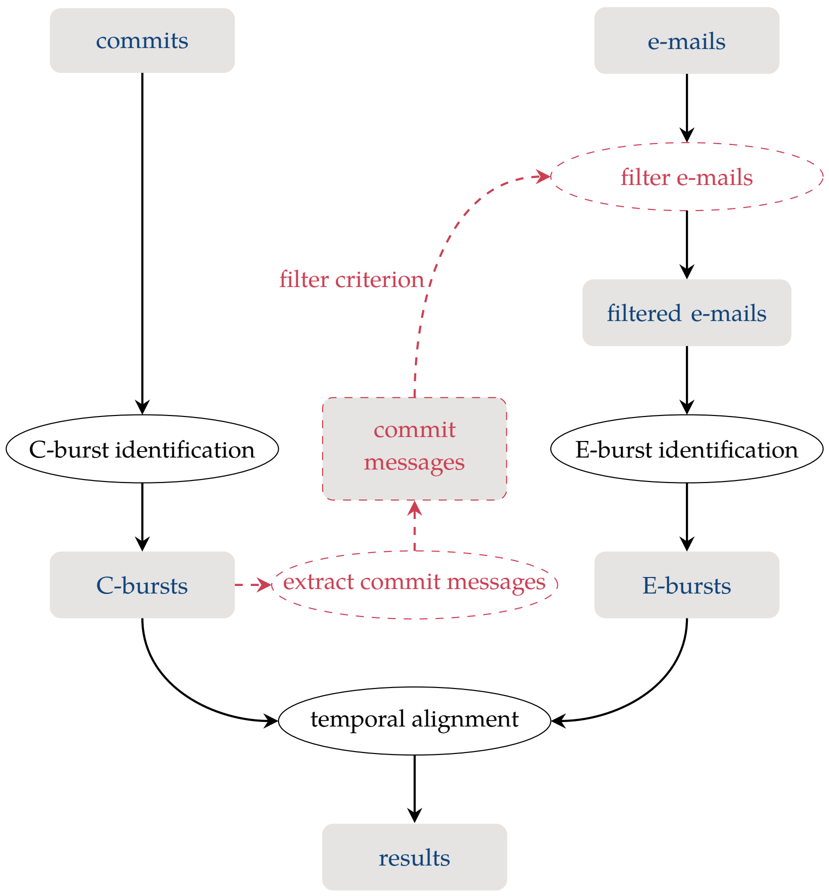


Figure 3.4: Workflow for one pair of developers using the upper-bound approach (black, solid arrows only) and the lower-bound approach (red, dashed, and black, solid arrows, together). Commit data are used to identify C-bursts. For the upper-bound approach, all e-mail data are used to identify E-bursts. For the lower-bound approach, the e-mails are first filtered based on the commit messages, which are extracted from the identified C-bursts of the developer pair. Only those e-mails whose subjects match one of the extracted commit messages are kept. Then, E-bursts are extracted from the filtered e-mails. In the end, independent of the approach, C-bursts and E-bursts of a developer pair are temporally aligned.

3.2.5 Upper-Bound and Lower-Bound Approach for Determining Coordination

To search for coordination between two developers, we check whether C-bursts and E-bursts of a developer pair are temporally aligned. However, we cannot be certain whether temporally aligned C-bursts and E-bursts are related to each other or whether they are completely unrelated and just are temporally aligned by coincidence. We elaborate on this later in detail and manually check in Section 3.5 for a small sample of our data whether and for which percentage of the E-bursts such relationships exist. As it is prohibitively time-consuming to manually decide for each pair of temporally aligned C-bursts and E-bursts whether they are related, we use two automatic approaches here, which we illustrate in Figure 3.4:

There is an upper bound for coordination, that is, we assume that all the temporally aligned C-bursts and E-bursts are content-wise related and therefore represent coordination (see Algorithm 3.2 for the trivial e-mail filtering in the upper-bound approach).

Algorithm 3.2 E-mail filtering in the upper-bound approach

```
Input: list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\}}$
1: ▷ no filtering needs to be performed
2: $\overline{e}_{\{A,B\},\text{filtered}} \leftarrow \overline{e}_{\{A,B\}}$
Output: list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\},\text{filtered}}$
```

Algorithm 3.3 E-mail filtering in the lower-bound approach

```
Input: list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\}}$,
       list of C-bursts of the developer pair $\{A,B\}$ $\overline{cb}_{\{A,B\}}$
1: $\overline{e}_{\{A,B\},\text{filtered}} \leftarrow \emptyset$
2: for each C-burst $cb \in \overline{cb}_{\{A,B\}}$ do
3:    $\overline{m}_{cb} \leftarrow$ extract commit messages from all commits belonging to $cb$
4:    for each e-mail $e \in \overline{e}_{\{A,B\}}$ do
5:       $s_e \leftarrow$ extract subject from e-mail $e$
6:       $s_e \leftarrow$ remove auto-generated prefixes like 'Re:' or 'Fwd:' or '[PATCH]' from $s_e$
7:       for each commit message $m \in \overline{m}_{cb}$ do
8:          if $m$ starts with $s_e$ then
9:             $\overline{e}_{\{A,B\},\text{filtered}} \leftarrow \overline{e}_{\{A,B\},\text{filtered}} \cup e$
10:            break
11:         end if
12:      end for
13:   end for
14: end for
Output: filtered list of e-mails of a developer pair $\{A,B\}$ $\overline{e}_{\{A,B\},\text{filtered}}$
```

Alternatively, in many OSS projects, there is information on the relation between e-mails and commits. For example, if code changes (which form a commit) have to be submitted to the mailing list in form of a patch (like in QEMU), the e-mail subject is often automatically generated out of the heading of the commit message. This way, we can learn that contents of e-mail threads whose subject is also the beginning of a commit message are related to the respective commit. Therefore, temporally aligned C-bursts and E-bursts for which one e-mail of the E-burst has a subject which belongs to a commit message of the temporally aligned C-burst are content-wise related and, hence, indicate coordination. Nevertheless, this might not be the only kind of coordination as e-mails that do not follow this convention could also contain content-wise related information. This is why we call the approach lower-bound approach (see Algorithm 3.3 for the e-mail filtering based on commit messages in the lower-bound approach). E-mail filtering in the lower-bound approach has a complexity of $O(|\overline{D}|^2 \cdot |\overline{c_{max}}| \cdot |\overline{e_{max}}|)$, with $|\overline{D}|$ being the number of developers, $|\overline{c_{max}}|$ being the maximum number of commits of a single developer, and $|\overline{e_{max}}|$ being the maximum number of e-mails of a single developer in the project.

Both the upper-bound and the lower-bound approach will not represent the actual amount of coordination, but by using an upper-bound and a lower bound we are able to narrow down the problem and know that the truth must be somewhere in-between these bounds.

3.2.6 Time-Series Analysis of C-Curves and E-Curves

To check whether C-bursts and E-bursts of a developer pair are synchronized, we need to measure the similarity between both sequences of bursts. For this purpose, we construct C-curves and E-curves for each developer pair. The curves denote the number of commits and e-mails that are contained in a burst aggregated for each day of the time series. That is, we build a histogram of the numbers of these commits and e-mails per day and derive a curve from that, as depicted in Figure 3.2.

Since commit activities and e-mail activities rarely occur at the same instant of time, the comparison of C-curves and E-curves needs to be error-tolerant such that we are able to tolerate slight temporal shifts between C-bursts and E-bursts. For example, as developers need some time to create a commit and also some time passes until an e-mail is written, we aggregate the number of commits or e-mails belonging to a burst on a daily basis so that we can perceive bursts of developer activity more clearly. To incorporate also latent times of activity (such as time for implementation, testing, or planning) into this line of thought, we use rectangular smoothing, which achieves two objectives: (1) We reduce noise in the curves and alleviate the intensity of a burst at a specific day (as the aggregation on a daily basis is sensitive to the distribution of the commits or e-mails among several days), and (2) we slightly broaden bursts in the curve to cover that developers may prepare or deal with source-code changes or communication activity longer than the actual work on commits and e-mails lasts, to be robust to a shift of several days between C-bursts and E-bursts. We use a smoothing parameter of 2ξ (with ξ being the time-window parameter used for burst identification). That is, we also take ξ days before and after a burst into account to check whether C-bursts and E-bursts are synchronized. (We also tried other smoothing parameters, which led to similar results, though. For more information, please refer to the supplementary website.)

To compare C-curves and E-curves, we use dynamic time warping (DTW) [RJ93], effectively calculating their distances (as introduced in Section 3.1.3). The DTW algorithm transforms one time series (the query) into the other (the reference) and measures the transformation costs. The higher the resulting transformation costs, the greater the distance between the compared time series. In addition, we use a Sakoe-Chiba band [SC78] to constrain the maximum allowed time deviation between two matched data points. The Sakoe-Chiba band prohibits global deformations to match the time series restricting the optimization algorithm to only local transformation operations. With that, we have a global constraint that allows only close-by bursts of the two time series to get matched. Consequently, when using a band-window size of 2ξ, we restrict C-bursts and E-bursts to get matched when their data points have a maximal distance of 2ξ days.

The outcome of the DTW calculation is a distance value describing how different the C-burst time series and the corresponding E-burst time series are under the defined transformation restrictions. So, using the DTW, we can measure the temporal correlation between the C-bursts and the E-bursts of a developer pair.

Table 3.2: Analyzed time range and size (in terms of numbers of developers, commits, LOC, files, e-mails, and e-mail threads) of the subject projects.

		QEMU	BusyBox	OpenSSL
Analyzed time range	Start:	2003-02-18	2003-01-14	2002-02-18
	End:	2016-07-27	2016-02-19	2016-02-19
# developers		951	230	168
# developer pairs		451 725	26 335	14 028
Average # developers active per year		151	34	26
# commits		35 608	10 087	7 887
Average # commits per year		3 484	831	791
# LOC (at the latest analyzed commit)		1 106 794	229 087	334 149
# files		3 165	1 362	1 378
# e-mails (messages)		374 815	23 527	10 228
# e-mail threads (conversations)		52 170	7 320	6 280

3.3 Study Design

In our empirical study, we consider coordination in synchronous development on different abstraction levels, namely message-based communication on the mailing list and conversation-based communication on the mailing list. For this purpose, we analyze the OSS projects QEMU, BusyBox, and OpenSSL. In this section, we provide information on our subject projects, describe our data-extraction procedure, give a description of the experiment variables, and formulate our hypotheses.

3.3.1 Subject Projects

We analyze three different OSS projects: QEMU, BusyBox, and OpenSSL. QEMU is a virtual-machine emulator. Within the analyzed time range of more than 13 years, QEMU had 951 developers. QEMU has a policy[4] that forces developers to send patches to the mailing list first. BusyBox is a UNIX command-line tool suite, to which 230 developers contributed within the analyzed time range. OpenSSL is an encryption library to secure Internet connections, having 168 developers in the analyzed time range. For all projects, we analyze all commits and e-mails (sent by developers who also contributed to the source code) between 2002 and 2016. This sums up to about 54 000 commits and 409 000 e-mails across all projects. We provide more details and descriptive statistics for each project in Table 3.2. All projects used a mailing list as well-established and—in the time range we analyze—persistent contribution system to discuss patches and share developer knowledge. As the three projects differ in size, commit policies, and application domain, they already provide substantial insights into developer coordination. Although, due to high computation time and high memory consumption when identifying bursts, we cannot analyze more projects with reasonable effort.

4 `https://wiki.qemu.org/Contribute/SubmitAPatch/` (accessed at 2019-02-12)

Table 3.3: Independent and dependent variables of our empirical study.

Independent Variables	Dependent Variables
– Time window ξ (1 day, 5 days, 10 days, 15 days) – Abstraction of communication (message-based, conversation-based) – Filtering of the e-mails (none (upper-bound), by C-bursts' commit messages (lower-bound))	– Number of C-bursts – Synchronicity degree deg_{sync} – Classification of commits (synchronous and non-synchronous) – Code growth ΔL – Implementation effort ΔW – Number of E-bursts – DTW distances describing the temporal correlation between C-curves and E-curves

3.3.2 Data Extraction

For each of the subject projects, we use Codeface to gather commit data from the corresponding Git repository and e-mail data from the publicly available mailing-list archive Gmane, as explained in Section 2.4. We limit the extraction of commit data to files that are implementation-related, so header files, documentation files, and build files are not considered in our study. Also, we only downloaded the e-mails of the corresponding developer mailing list (not user mailing lists) for each subject project, as we focus on investigating the coordination of source-code changes of developers. In our study, we only include e-mails of developers who also appear in commits, because we want to investigate the relation between C-bursts and corresponding E-bursts.

3.3.3 Variables

In Table 3.3, we provide an overview of independent and dependent variables of our study.

As independent variables, we vary the time window for burst identification. We consider time windows ξ of 1 day, 5 days, 10 days, and 15 days, based on a response-time analysis. In Figure 3.5, we show the response times for subsequent e-mails and subsequent commits to a common artifact for each pair of developers, for each level of abstraction and subject project considered in our study. Depending on project, interaction channel (source-code files vs. mailing-list communication), and abstraction level of communication (conversations vs. messages), at least, 70% of the different response times are shorter than 15 days. Especially on mailing lists, at least, 70% of the response times are even shorter than 1 day. Hence, our chosen time windows ξ are reasonable time distances that synchronous development can deliberately last, because developers mostly reply to e-mails or commits within a few days.

For identifying E-bursts, we distinguish between message-based E-bursts and conversation-based E-bursts. On top of that, we also vary the e-mail filtering: Whereas we use all e-mails and perform no filtering for our upper-bound approach, we filter the e-mails by commit messages of the C-bursts before identifying E-bursts for the lower-bound approach.

The results of our study depend on the variations of the above described independent variables (see Table 3.3). In particular, the number of bursts, the synchronicity degree of C-bursts, the differentiation of commits into synchronous and non-synchronous, as well as code

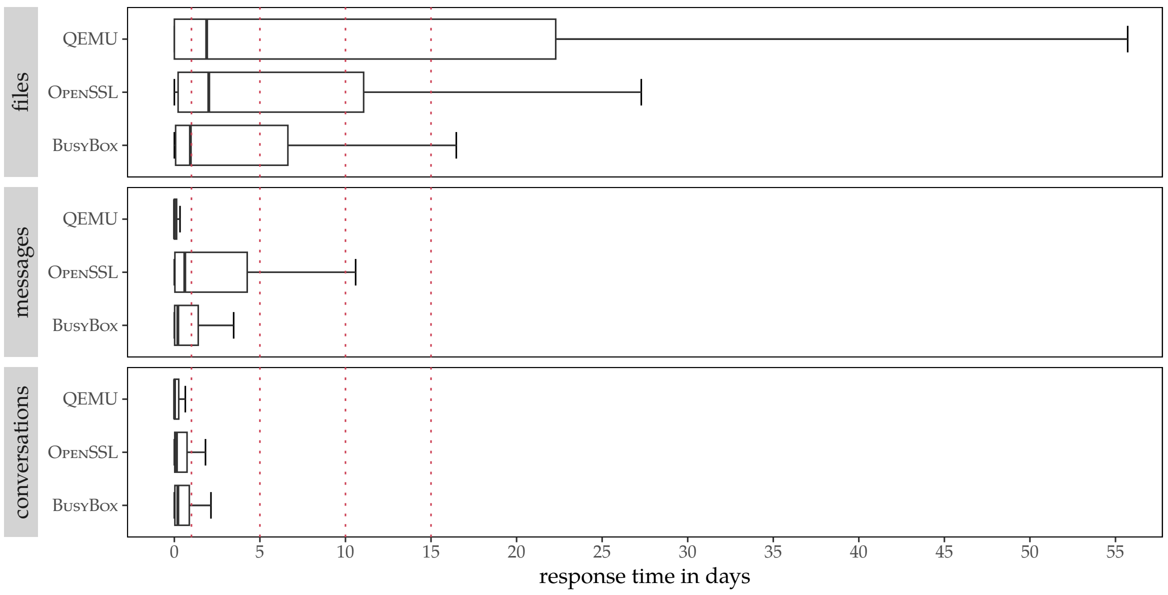


Figure 3.5: Box plots of the response times for each pair of developers; outliers are omitted. File-based response times represent the time distances between subsequent commits to a common file of a developer pair. Message-based response times represent the time distances between subsequent e-mails of a developer pair, conversation-based response times only represent the time distances between subsequent e-mails of a developer pair within the same thread. The red, dashed vertical lines represent the time windows ξ chosen for our study.

growth and implementation effort depend on the time window ξ. Notice that we consider a commit to be synchronous if, at least, one of its changed artifacts (i.e., files) is mutually changed within a C-burst. The number of E-bursts as well as the temporal correlation between C-bursts and E-bursts depend not only on the time window, but also on the abstraction of communication and on the e-mail filtering.

3.3.4 Null Model

To determine whether the bursts and their synchronicity degrees are just artifacts of a purely random process, we use a simulation technique based on synthetic datasets drawn from a null model. The null models, which represent random time series, allow us to test whether empirically observed bursts are significantly different from purely random bursts (i.e., whether they convey information). That is, by using a null model, we check whether our results are dependent on our variables or arise randomly. Specifically, we use the null models of the original study, as we explain next [XGD+12].

For commit data, we generate synthetic data based on a null model by purely randomizing the time intervals between two successive commits for each developer. The randomization operation is performed by randomly permuting the time intervals between all commits of the considered developer, as we describe in Algorithm 3.4. This way, for each developer, the distribution of the time intervals, the order of the commits, and the artifacts changed by this developer are preserved. C-bursts generated from the purely randomized time series are referred to as *simulated C-bursts*.

Algorithm 3.4 Generation of simulated C-bursts

```
Input: list of commits $\bar{c}$ (annotated with timestamps and developer names)
 1: ▷ generate simulated commit time series
 2: $\bar{c}_{sim} \leftarrow \emptyset$
 3: for each developer $D$ do
 4:     $\bar{c}_D \leftarrow$ commits in $\bar{c}$ authored by $D$
 5:     $t(D) \leftarrow$ sorted list of timestamps of $\bar{c}_D$
 6:
 7:     ▷ create an ordered list intervals to store the lengths
 8:     ▷ of intervals between two subsequent commits of $D$
 9:     for each $i \in 2..\text{length}(t(D))$ do
10:         $interval \leftarrow t(D)_i - t(D)_{i-1}$
11:         $intervals_i \leftarrow interval$
12:     end for
13:
14:     ▷ randomize the order of the intervals, but keep the same distribution
15:     $randomizedIntervals \leftarrow$ shuffle the elements in $intervals$
16:
17:     ▷ generate simulated list of timestamps $s(D)$
18:     $s(D)_1 \leftarrow t(D)_1$
19:     for each $i \in 2..\text{length}(t(D))$ do
20:         $s(D)_i \leftarrow s(D)_{i-1} + randomizedIntervals_{i-1}$
21:     end for
22:     $\bar{c}_{D,sim} \leftarrow$ update the timestamps in the list of commits $\bar{c}_D$ according to $s(D)$
23:     $\bar{c}_{sim} \leftarrow \bar{c}_{sim} \cup \bar{c}_{D,sim}$
24: end for
25:
26: ▷ extract simulated C-bursts
27: apply Algorithm 3.1 to the simulated commit list $\bar{c}_{sim}$
Output: simulated C-bursts for each pair of developers
```

For e-mail data, we use a similar approach. The only difference is that we do not randomize the time intervals between the e-mails of each developer, but the time intervals between successive e-mails of each pair of developers to preserve the order of e-mails sent by two different developers. So, each pair of developers has their own simulated e-mail time series, as we describe in Algorithm 3.5. E-bursts of the purely randomized time series are referred to as *simulated E-bursts*.

Generating a simulated commit time series (without burst detection) has a complexity of $\mathcal{O}(|\overline{D}| \cdot |\overline{c_{max}}|)$, and generating a simulated e-mail time series has a complexity of $\mathcal{O}(|\overline{D}|^2 \cdot |\overline{e_{max}}|^2)$, with $|\overline{D}|$ being the number of developers in the project, $|\overline{c_{max}}|$ being the maximum number of commits of a single developer in the project, and $|\overline{e_{max}}|$ being the maximum number of e-mails of a single developer in the project.

For both, commit and e-mail time series, we generate 100 simulated time series each, except for one subject project. For QEMU, we only generate 2 simulations of the e-mail time series due to computational limitations.[5] See Section 3.7 for a discussion of threats to validity.

[5] To identify E-bursts in 100 simulations of QEMU for four different time windows and two abstraction levels, we would need about 22 months using 40 nodes, having a 2.2 GHz processor, with 20 cores each in parallel and 128 GB RAM per node.

Algorithm 3.5 Generation of simulated E-bursts

```
Input: list of e-mails $\bar{e}$ (annotated with timestamps and developer names)
 1: ▷ generate a simulated e-mail time series for each pair of developers
 2: for each pair of developers $\{A,B\}$ do
 3:     $\bar{e}_{\{A,B\}} \leftarrow$ e-mails in $\bar{e}$ sent by $A$ or $B$
 4:     $t(A,B) \leftarrow$ sorted list of timestamps of $\bar{e}_{\{A,B\}}$
 5:
 6:     ▷ create an ordered list intervals to store the lengths
 7:     ▷ of intervals between two subsequent e-mails sent by $A$ or $B$
 8:     for each $i \in 2..\text{length}(t(A,B))$ do
 9:         $interval \leftarrow t(A,B)_i - t(A,B)_{i-1}$
10:         $intervals_i \leftarrow interval$
11:     end for
12:
13:     ▷ randomize the order of the intervals, but keep the same distribution
14:     $randomizedIntervals \leftarrow$ shuffle the elements in $intervals$
15:
16:     ▷ generate simulated list of timestamps $s(A,B)$
17:     $s(A,B)_1 \leftarrow t(A,B)_1$
18:     for each $i \in 2..\text{length}(t(A,B))$ do
19:         $s(A,B)_i \leftarrow s(A,B)_{i-1} + randomizedIntervals_{i-1}$
20:     end for
21:     $\bar{e}_{\{A,B\},sim} \leftarrow$ update the timestamps in the list of e-mails $\bar{e}_{\{A,B\}}$ using $s(A,B)$
22:
23:     ▷ extract simulated E-bursts
24:     apply E-burst extraction to pair $\{A,B\}$ using $\bar{e}_{\{A,B\},sim}$, as described in Section 3.1.1
25: end for
Output: simulated E-bursts for each pair of developers
```

3.3.5 Hypotheses

Next, we introduce our hypotheses. Each hypothesis is evaluated by varying time window (1 day, 5 days, 10 days, 15 days) and, when related to E-bursts, by varying the abstraction level of coordination (message-based communication vs. conversation-based communication).

Before introducing our hypotheses, let us explain our numbering scheme for hypotheses: Our main hypotheses (i.e., H2 and H3) are related to the comparison of the different abstraction levels of communication. However, before comparing abstraction levels, we first check whether the underlying hypotheses of the original study hold. To check that the occurrence of C-bursts is not a purely random process, we introduce the four corresponding sub-hypotheses H1.1, H1.2, H1.3, and H1.4. Moreover, with regard to the comparison of the different abstraction levels of coordination, we introduce the underlying hypotheses of the original study as sub-hypotheses of our main hypotheses (i.e., H2.1, and H3.1). To answer our main hypotheses (i.e., to compare the different abstraction levels), we compare the outcomes of the corresponding sub-hypotheses on the different abstraction levels. So, for comparing abstraction levels, we lift the particular sub-hypotheses and define composed hypotheses (i.e., H2+H2.1, and H3+H3.1). In Figure 3.6, we provide an overview of all (sub-)hypotheses.

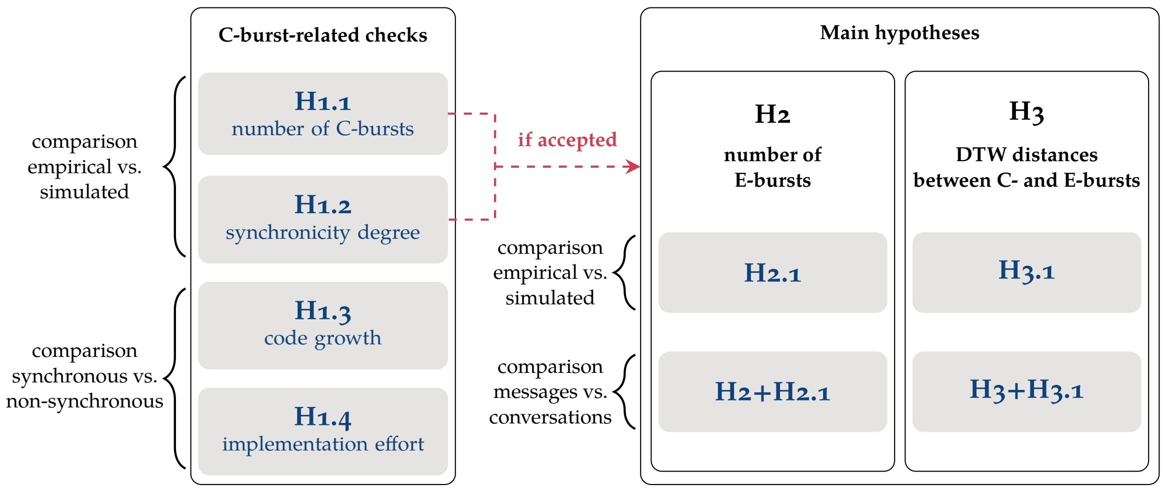

C-burst-related checks
H1.1
number of C-bursts
H1.2
synchronicity degree
H1.3
code growth
H1.4
implementation effort
comparison empirical vs. simulated
comparison synchronous vs. non-synchronous
if accepted
Main hypotheses
H2
number of E-bursts
H3
DTW distances between C- and E-bursts
comparison empirical vs. simulated
comparison messages vs. conversations
H2.1
H3.1
H2+H2.1
H3+H3.1

Figure 3.6: Overview of our hypotheses and sub-hypotheses.

Hypotheses Related to C-Bursts First, we define hypotheses regarding the collaboration of developers. To examine that developer collaboration is not a purely random process and has an effect on the number of C-bursts and their characteristics, we formulate the following four hypotheses:

H1.1: The number of empirical C-bursts is higher than the number of simulated C-bursts.

H1.2: The synchronicity degree of empirical C-bursts is higher than the one of simulated C-bursts.

H1.3: The code growth ΔL is higher in synchronous commits than in non-synchronous commits.

H1.4: The implementation effort ΔW is lower in synchronous commits than in non-synchronous commits.

All of the above hypotheses are derived from the the original study. Nevertheless, since code growth and implementation effort might be highly project-dependent measures, H1.3 and H1.4 do not necessarily need to be fulfilled even if developer collaboration does not follow a purely random process. Therefore, and in contrast to H1.1 and H1.2, we added H1.3 and H1.4 here only for the sake of completeness with respect to the original study.

Hypotheses Related to E-Bursts Second, as developers converse via mailing lists, we formulate hypotheses regarding which abstraction level of mailing-list communication captures coordination of developers best. That is, we expect that conversations capture the coordination activity among developers more accurately than considering individual messages only, as e-mail conversations represent the conceptual relationship between e-mails.

H2: Conversation-based communication captures developer coordination more accurately than message-based communication.

In particular, we check the following sub-hypothesis for both conversation-based and message-based communication, as a high amount of collaboration activity should be aligned with a high amount of coordination:

H2.1: The relation between the number of C-bursts and the number of E-bursts is described by a linear relationship.

For this hypothesis, we consider only developer pairs that have, at least, one C-burst and, at least, one E-burst, since we aim at analyzing developer pairs that contribute to the source code *and* communicate on the mailing list. Here, we use both our upper-bound and our lower-bound approach and evaluate the hypothesis separately for both approaches. After checking this sub-hypothesis for all abstraction levels separately, we compare the strengths of the linear relationships of message-based and conversation-based coordination to answer H2 conclusively:

H2+H2.1: The linear relation between the number of C-bursts and the number of E-bursts has a higher goodness of fit for conversation-based E-bursts than for message-based E-bursts.

Here, we evaluate the hypothesis separately for the upper-bound approach and for the lower-bound approach.

Hypotheses Related to C-Bursts and E-Bursts Finally, we investigate the temporal relationship between C-bursts and E-bursts of developer pairs with the following hypothesis:

H3: The temporal correlation between C-bursts and E-bursts is higher for conversation-based E-bursts than for message-based E-bursts.

If a C-burst and an E-burst of the same developer pair are temporally related, this is an indicator of a relationship between these bursts. That is, an E-burst that appears right before or after a C-burst may address the discussion of the code changes applied in the C-burst. To answer H3, we check the following sub-hypothesis for each abstraction level of mailing-list communication to examine that the empirical DTW distances are not purely random and empirical C-bursts and E-bursts are dependent processes:

H3.1: C-bursts and E-bursts are temporally correlated, that is, the DTW distances between empirical C-curves and empirical E-curves are smaller than between simulated curves.

That is, we expect related C-bursts and E-bursts to appear temporally close to each other, resulting in smaller DTW distances than for simulated bursts. Again, we analyze the temporal correlation only for developer pairs that have, at least, one C-burst and, at least, one E-burst.

After checking the sub-hypothesis, we compare the empirical DTW distances of the different abstraction levels with each other to answer H3 conclusively:

H3+H3.1: The DTW distances between C-curves and E-curves are smaller for conversation-based E-bursts than for message-based E-bursts.

We use the E-bursts of our two different approaches separately and evaluate our hypothesis for both the upper-bound and the lower-bound approach.

Statistical Tests To test the hypotheses, we use a suite of statistical methods. All the above stated hypotheses are alternative hypotheses.

For H1.1, we use a one-tailed, paired Wilcoxon signed-rank test to compare the numbers of empirical and simulated C-bursts for each pair of developers. Therefore, for each of the developer pairs, we compute the median of the numbers of C-bursts of the 100 simulations and compare this median with the empirical number of C-bursts of the developer pair. (We use an aggregated measure of the 100 simulations to be able to use a paired test for comparing the real and empirical numbers of C-bursts per developer pair. To be robust to outliers, we use the median.) Here, we consider also developer pairs that have no C-burst at all, since the number of developer pairs having no burst can be different and, therefore, can affect the comparison. In addition to the Wilcoxon signed-rank test, we compute the corresponding effect size r.[6]

For H1.2, H1.3, and H1.4, we use a one-tailed, unpaired Mann-Whitney U test: We compare the whole population of the synchronicity degrees of all empirical bursts with the whole population of the synchronicity degrees of all 100 simulations together. We also compute Cliff's Delta, which quantifies the effect size that corresponds to the Mann-Whitney U test.

For H2 and H2.1, we fit a linear regression model and compare the fitted models by comparing their adjusted R-square values and p-values.

For H3.1, we use a one-tailed, unpaired Mann-Whitney U test to compare the empirical DTW distances of all considered developer pairs with the simulated DTW distances of all simulations together. Note that we use a Mann-Whitney U test because the number of available data points can be rather small (especially when using the lower-bound approach, we have only few developer pairs that have, at least, one C-burst and, at least, one E-burst in some cases), and the data are not necessarily normally distributed, which we measured using the Shapiro-Wilk test. Corresponding to the Mann-Whitney U test, we again use Cliff's Delta to quantify the effect size.

Finally, for the comparison of the different abstraction levels of mailing-list communication in hypothesis H3, we use a Mann-Whitney U test in combination with false-discovery rate (FDR) correction to account for multiple testing.

3.4 Results

Next, we present our results.[7] To generate the results (including simulations), we used the following hardware in parallel over several weeks: 40 computation nodes having a 2.2 GHz processor, with 20 cores each and 128 GB RAM per node.

3.4.1 C-Bursts

As we show in Table 3.4, we see that the file-based C-bursts per developer pair occur significantly more frequently than purely by chance ($p < 0.05$), for all subjects projects, independent of time window ξ. However, the corresponding effect size is very low (absolute values be-

6 `https://www.rdocumentation.org/packages/rcompanion/versions/2.3.7/topics/wilcoxonPairedR/` (accessed at 2019-02-12)

7 When we state that we accept a certain hypothesis (all our hypotheses are alternative hypotheses, as stated above), we actually mean that we reject the corresponding null hypothesis.

Table 3.4: Paired, one-tailed Wilcoxon signed-rank test for comparing empirical simulated numbers of C-bursts per developer pair **(H1.1)**. (We use the median of all simulations to get one value per developer here.) s denotes the standard deviation. V represents the V-statistic of the Wilcoxon signed-rank test, the corresponding p-value indicates whether the alternative hypothesis H1.1 is accepted ($p < 0.05$) or not. r denotes the effect size corresponding to the paired Wilcoxon signed-rank test.

		# Bursts (mean ± s)				
Project	ξ	Empirical	Simulated	V	p-value	r
QEMU	1	0.005 ± 0.117	0.002 ± 0.079	706 200	< **0.05**	−0.041
	5	0.011 ± 0.235	0.007 ± 0.213	2 948 700	< **0.05**	−0.045
	10	0.017 ± 0.305	0.012 ± 0.280	5 925 700	< **0.05**	−0.047
	15	0.022 ± 0.340	0.015 ± 0.310	8 491 000	< **0.05**	−0.048
BusyBox	1	0.012 ± 0.345	0.003 ± 0.132	6 213	< **0.05**	−0.053
	5	0.018 ± 0.482	0.009 ± 0.327	9 034	< **0.05**	−0.050
	10	0.020 ± 0.467	0.012 ± 0.360	11 769	< **0.05**	−0.045
	15	0.020 ± 0.405	0.014 ± 0.347	13 247	< **0.05**	−0.038
OpenSSL	1	0.027 ± 0.574	0.008 ± 0.197	4 983	< **0.05**	−0.069
	5	0.057 ± 0.963	0.029 ± 0.583	21 636	< **0.05**	−0.090
	10	0.067 ± 1.046	0.047 ± 0.780	24 502	< **0.05**	−0.066
	15	0.071 ± 1.035	0.055 ± 0.859	26 400	< **0.05**	−0.055

Table 3.5: Paired, one-tailed Wilcoxon signed-rank test for comparing empirical simulated numbers of C-bursts per developer pair **(H1.1)** **only for developer pairs that have, at least, one C-burst**. (We use the median of all simulations to get one value per developer here.) s denotes the standard deviation. V represents the V-statistic of the Wilcoxon signed-rank test, the corresponding p-value indicates whether the alternative hypothesis H1.1 is accepted ($p < 0.05$) or not. r denotes the effect size corresponding to the paired Wilcoxon signed-rank test.

		# Bursts (mean ± s)				
Project	ξ	Empirical	Simulated	V	p-value	r
QEMU	1	1.578 ± 1.515	0.474 ± 1.353	660 460	< **0.05**	−0.822
	5	1.864 ± 2.352	0.987 ± 2.488	2 533 500	< **0.05**	−0.705
	10	1.901 ± 2.559	1.126 ± 2.652	4 927 400	< **0.05**	−0.659
	15	1.889 ± 2.560	1.168 ± 2.612	6 903 100	< **0.05**	−0.642
BusyBox	1	2.802 ± 4.514	0.613 ± 1.918	5 974.0	< **0.05**	−0.857
	5	3.207 ± 5.651	1.545 ± 4.092	8 276.0	< **0.05**	−0.758
	10	2.994 ± 4.952	1.712 ± 4.088	9 845.5	< **0.05**	−0.672
	15	2.628 ± 3.894	1.635 ± 3.642	9 721.5	< **0.05**	−0.599
OpenSSL	1	3.519 ± 5.595	0.934 ± 2.042	4 465	< **0.05**	−0.841
	5	3.299 ± 6.580	1.620 ± 4.133	19 662	< **0.05**	−0.767
	10	3.019 ± 6.381	1.930 ± 4.864	19 694	< **0.05**	−0.614
	15	2.766 ± 5.863	1.981 ± 4.973	20 038	< **0.05**	−0.522

Table 3.6: One-tailed Mann-Whitney U test for comparing synchronicity degrees (**H1.2**). s denotes the standard deviation. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the alternative hypothesis H1.2 is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		deg_{sync} (mean $\pm s$)				
Project	ξ	Empirical	Simulated	U	p-value	Cliff's Delta
QEMU	1	0.466 ± 0.352	0.340 ± 0.306	169 020 000	$< \mathbf{0.05}$	0.207
	5	0.369 ± 0.325	0.296 ± 0.284	1 403 200 000	$< \mathbf{0.05}$	0.123
	10	0.332 ± 0.308	0.270 ± 0.269	3 296 700 000	$< \mathbf{0.05}$	0.110
	15	0.307 ± 0.297	0.254 ± 0.260	5 078 600 000	$< \mathbf{0.05}$	0.097
BusyBox	1	0.605 ± 0.354	0.320 ± 0.300	2 416 800	$< \mathbf{0.05}$	0.450
	5	0.470 ± 0.346	0.248 ± 0.249	9 497 500	$< \mathbf{0.05}$	0.387
	10	0.434 ± 0.341	0.233 ± 0.243	13 510 000	$< \mathbf{0.05}$	0.357
	15	0.411 ± 0.339	0.225 ± 0.239	14 568 000	$< \mathbf{0.05}$	0.326
OpenSSL	1	0.423 ± 0.336	0.302 ± 0.284	3 616 500	$< \mathbf{0.05}$	0.209
	5	0.342 ± 0.299	0.248 ± 0.241	24 001 000	$< \mathbf{0.05}$	0.180
	10	0.316 ± 0.286	0.227 ± 0.226	40 996 000	$< \mathbf{0.05}$	0.183
	15	0.304 ± 0.277	0.220 ± 0.221	51 711 000	$< \mathbf{0.05}$	0.183

tween 0.03 and 0.09). The reason for the low effect size is that most of the developer pairs have no C-burst at all. For example, in QEMU, only ~1% of the developer pairs have, at least, one C-burst, due to the combinatorial explosion of developer pairs. Nevertheless, when we restrict our analysis to developer pairs that have, at least, one C-burst, then we still get significant results (empirical C-bursts occur more frequently than purely by chance), but we get higher effect sizes (absolute values between 0.52 and 0.86, see Table 3.5). Regardless of that, also the overall number of empirical C-bursts is higher than the overall number of simulated C-bursts (using the median of the 100 simulations to get one number per developer pair). For example, for QEMU with $\xi = 5$, there are 5 185 empirical C-bursts, but only 3 122 simulated ones. There are similar results for BusyBox and OpenSSL. Consequently, *we accept H1.1.*

Also, the synchronicity degrees of the empirical C-bursts are significantly higher than the synchronicity degrees of the simulated C-bursts (see Table 3.6; $p < 0.05$) for all abstraction levels and time windows. This holds for each subject project; for BusyBox and OpenSSL the corresponding effect sizes are even higher than for QEMU. Hence, *we accept H1.2.*

Next, we have a look at the code growth in synchronous and non-synchronous commits. In Table 3.7, we observe that, according to the Mann-Whitney U test ($p < 0.05$), only for QEMU with $\xi = 1$ or $\xi = 5$ the code growth (ΔL) is higher in synchronous commits than in non-synchronous commits. For the other time windows of QEMU and also for the other projects, the code growth in synchronous commits is not higher than the code growth in non-synchronous commits. As we can observe a higher code growth in synchronous commits than in non-synchronous commits only in some rare cases, but not in general, *we reject H1.3.*

When we look at the implementation effort (see Table 3.8), we find that ΔW is not lower in synchronous commits than in non-synchronous commits, in all cases for all three projects. Hence, as there is no statistically significant difference, *we reject H1.4.*

Table 3.7: One-tailed Mann-Whitney U test for comparing the code growth ΔL of synchronous commits and non-synchronous commits (**H1.3**). s denotes the standard deviation. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the alternative hypothesis H1.3 is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		ΔL (mean $\pm$ s)				
Project	ξ	Synchronous	Non-synchronous	U	p-value	Cliff's Delta
QEMU	1	42.647 ± 732.059	39.235 ± 263.358	96 963 000	< **0.05**	0.059
	5	34.986 ± 515.489	42.769 ± 285.679	152 310 000	< **0.05**	0.023
	10	34.232 ± 448.141	46.100 ± 308.759	158 750 000	0.22	0.005
	15	32.979 ± 415.286	51.160 ± 339.906	147 620 000	0.91	−0.009
BusyBox	1	34.277 ± 210.011	47.879 ± 683.844	4 371 200	0.34	0.008
	5	47.072 ± 546.765	46.474 ± 679.660	8 416 800	0.52	−0.001
	10	45.972 ± 497.104	46.865 ± 710.348	10 410 000	0.96	−0.022
	15	54.172 ± 688.270	42.260 ± 633.610	11 448 000	0.99	−0.028
OpenSSL	1	68.612 ± 578.951	64.724 ± 1 258.090	3 095 900	0.90	−0.026
	5	57.875 ± 418.904	68.003 ± 1 388.579	6 277 300	0.41	0.003
	10	111.117 ± 1 941.316	37.665 ± 203.151	7 286 000	0.48	0.001
	15	98.601 ± 1 781.892	38.369 ± 209.825	7 679 800	0.51	−0.000

Table 3.8: One-tailed Mann-Whitney U test for comparing the implementation effort ΔW of synchronous commits and non-synchronous commits (**H1.4**). s denotes the standard deviation. U represents the U-statistic of the Mann-Whitney U test, the corresponding p-value indicates whether the alternative hypothesis H1.4 is accepted ($p < 0.05$) or not. Cliff's Delta denotes the corresponding effect size.

		ΔW (mean $\pm$ s)				
Project	ξ	Synchronous	Non-synchronous	U	p-value	Cliff's Delta
QEMU	1	87.903 ± 764.969	74.342 ± 318.517	99 455 000	1.00	0.086
	5	78.643 ± 563.798	75.551 ± 326.277	158 850 000	1.00	0.067
	10	72.276 ± 493.614	78.330 ± 349.167	166 410 000	1.00	0.053
	15	72.550 ± 460.570	83.605 ± 378.551	155 570 000	1.00	0.045
BusyBox	1	127.371 ± 362.982	140.350 ± 982.852	4 655 300	1.00	0.074
	5	165.359 ± 1 366.644	132.183 ± 792.284	9 114 400	1.00	0.082
	10	162.381 ± 1 179.108	129.249 ± 820.708	11 488 000	1.00	0.079
	15	172.254 ± 1 222.082	120.152 ± 734.427	12 707 000	1.00	0.079
OpenSSL	1	743.601 ± 16 919.798	107.340 ± 1 326.779	3 667 200	1.00	0.153
	5	382.264 ± 10 903.567	102.872 ± 1 432.185	7 201 600	1.00	0.151
	10	369.613 ± 9 619.071	67.960 ± 242.243	8 425 100	1.00	0.157
	15	323.459 ± 8 825.309	66.634 ± 242.605	8 851 400	1.00	0.152

Overall, we conclude that the empirically observed C-bursts are significantly different from purely random C-bursts, as the empirically observed C-bursts occur more frequently (H1.1) and with a higher synchronicity degree (H1.2). Regarding code growth (H1.3) and implementation effort (H1.4), we do not observe any significant differences between synchronous and non-synchronous commits.

3.4.2 E-Bursts

For the comparison of message-based and conversation-based E-bursts, we first test for H2.1 whether the number of C-bursts and the number of E-bursts for a developer pair (having, at least, one burst each) are linearly dependent. We do this separately for the E-bursts extracted with our lower-bound approach and with our upper-bound approach, respectively. In the plots that follow, we present the results of our largest and, therefore, most representative subject project QEMU, for $\xi = 5$, which also exemplifies the results of the other time windows.[8]

Using the upper-bound approach, fitting a linear model on the message-based E-bursts results in a very small adjusted R-square value of 0.06 (see left plot in Figure 3.7), that is, only 6% of the variance is described by the linear model. Only a high adjusted R-square value would indicate that the model describes the data points well. In contrast, still using the upper-bound approach, the linear model of conversation-based E-bursts and C-bursts fits significantly better, as the adjusted R-square value of 0.55 in Figure 3.7 (right) illustrates. That is, 55% of the variance is described by the linear model, and hence there is a stronger linear relationship between the number of C-bursts and the number of E-bursts for conversation-based E-bursts than for message-based E-bursts.

Switching to the lower-bound approach, we get significant linear models for both message-based and conversation-based E-bursts, as the adjusted R-square values of 0.71 (message-based) and 0.37 (conversation-based) indicate (see Figure 3.8).

This result holds only for QEMU, though. In BusyBox and OpenSSL, there are too few data points to fit a significant linear model, especially in the lower-bound approach. Instead of fitting a linear model, we also tried to compute Spearman's rank correlation, which also led to similar, non-significant results for these subject projects.

Based on our results, *we accept H2.1 for conversation-based E-bursts*, as we obtain significant linear models for both the upper-bound and the lower-bound approach for project QEMU. However, *we are inconclusive regarding H2.1 for message-based E-bursts*, as for these we obtain a significant linear model only for the lower-bound approach.

Eventually, comparing the results for message-based and conversation-based E-bursts shows that conversation-based E-bursts have a better linear relationship to the number of C-bursts than message-based E-bursts for most of the cases when we use the upper-bound approach (see adjusted R-square values in Figure 3.7). Nevertheless, when we use the lower-bound approach, it is the other way round (see adjusted R-square values in Figure 3.8). So, overall, *we are inconclusive regarding H2* (see Table 3.9).

8 The corresponding plots for the other time windows and also for the other subject projects are available on our supplementary website: `https://se-sic.github.io/paper-coordination-bursts/`

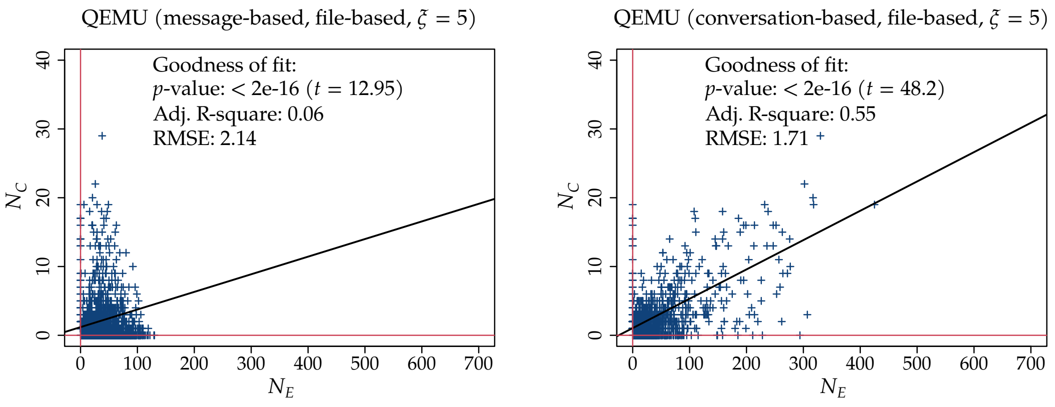


Figure 3.7: Upper-bound approach: Linear model fitting for the relationship between the number of C-bursts (N_C) and the number of message-based E-bursts (N_E) (left) and conversation-based E-bursts (right). Every data point represents one developer pair. Model fitting was applied only to developer pairs having, at least, one C-burst and, at least, one E-burst (data points above and right, respectively, of the red horizontal and vertical lines, respectively).

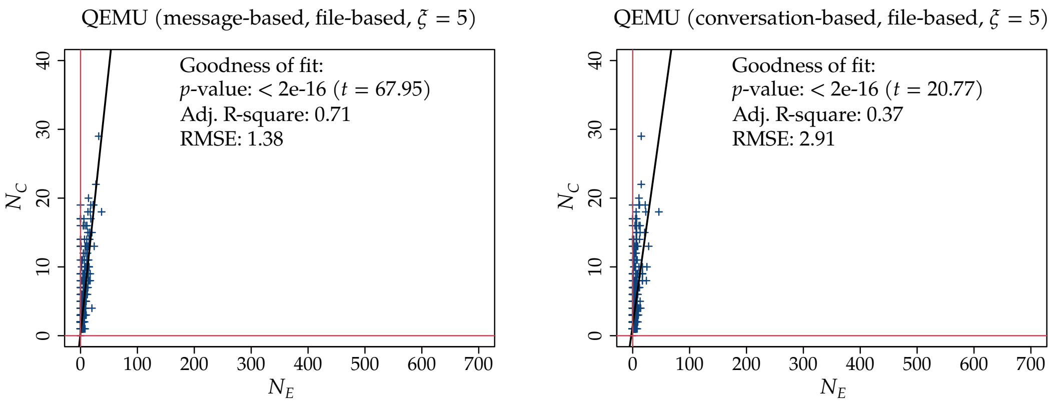


Figure 3.8: Lower-bound approach: Linear model fitting for the relationship between the number of C-bursts (N_C) and the number of message-based E-bursts (N_E) (left) and conversation-based E-bursts (right). Every data point represents one developer pair. Model fitting was applied only to developer pairs having, at least, one C-burst and, at least, one E-burst (data points above and right, respectively, of the red horizontal and vertical lines, respectively).

Table 3.9: Overview of the results regarding **H2** and its sub-hypothesis. ✓ denotes that we accept a (sub-)hypothesis, ✗ the denotes that we reject a (sub-)hypothesis, and ? denotes that we are inconclusive.

	Lower-Bound Approach	Upper-Bound Approach	H2.1
Messages	✓	✗	?
Conversations	✓	✓	✓
	Lower-Bound Approach	**Upper-Bound Approach**	**H2+H2.1**
Messages vs. conversations	✗	✓	?

Table 3.10: The results of testing **H3.1** for different abstraction levels of coordination for the upper-bound approach and for the lower-bound approach. ✓ denotes that the ***empirical*** DTW distances are ***smaller than*** the corresponding ***simulated*** ones, ✗ the opposite. ? denotes that there are insufficient data (no E-bursts at all).

	ξ	QEMU		BusyBox		OpenSSL	
		Messages	Conversations	Messages	Conversations	Messages	Conversations
Upper-bound approach	1	✓	✓	✓	✓	✓	✗
	5	✓	✓	✓	✓	✓	✓
	10	✓	✓	✓	✓	✓	✓
	15	✓	✓	✓	✓	✓	✓
Lower-bound approach	1	✓	✓	✓	✓	✗	?
	5	✓	✓	✓	✓	✗	✗
	10	✓	✓	✓	✓	✗	✗
	15	✓	✓	✓	✓	✓	✗

3.4.3 Temporal Correlation Between C-Bursts and E-Bursts

Finally, we search for a temporal correlation between C-bursts and E-bursts by computing the DTW distances of the C-curves and E-curves of each developer pair.

First of all, we investigate whether the temporal correlation between empirical C-bursts and empirical E-bursts is different from the temporal correlation between C-bursts and E-bursts in the null model. We state the corresponding results in Table 3.10, for both the upper-bound approach and the lower-bound approach. First, we have a look at the upper-bound approach: For QEMU, BusyBox, and OpenSSL, message-based E-bursts have significantly smaller DTW distances to C-bursts than in the null model, for all time windows. For conversation-based E-bursts, we also obtain significantly smaller DTW distances to the C-bursts than in the null model, except for OpenSSL with $\xi = 1$. When we use the lower-bound approach, also in all cases of QEMU and BusyBox the empirical DTW distances are smaller than the simulated ones. For OpenSSL, this holds only for message-based E-bursts with $\xi = 15$. The reason for that is that there are too few data points: When using the lower-bound approach, we often have zero E-bursts per developer pair in OpenSSL, resulting in empty DTW curves. Taken all together, when we neglect the cases where we have insufficient or few data, the empirical DTW distances are (with few exceptions) shorter for empirical bursts than for the null model, for both the upper-bound and the lower-bound approach. Consequently, *we accept H3.1 for message-based and also for conversation-based E-bursts.*

After comparing the empirical DTW distances with our null models, we now compare the outcomes of the different abstraction levels of the empirical data with each other. As we can see in Table 3.11, in which we provide an overview of the corresponding results for the comparison of message-based E-bursts and conversation-based E-bursts, we obtain a complex picture for H3+H3.1: When analyzing QEMU with the upper-bound approach, using conversation-based E-bursts leads to significantly lower DTW distances than using message-based E-bursts, independent of ξ. However, when using the lower-bound approach, this does not hold at all. Also for BusyBox, no matter which approach is used, this does not

Table 3.11: The results of testing **H3+H3.1** for comparing message and conversation level using lower-bound or upper-bound approach, respectively. ✓ denotes that the DTW distances using ***conversation-based*** E-bursts are ***smaller than*** the DTW distances using ***message-based*** E-bursts, ✗ the opposite. ? denotes that there are insufficient data (no E-bursts at all).

	ξ	QEMU	BusyBox	OpenSSL
Upper-bound approach	1	✓	✗	✓
	5	✓	✓	✗
	10	✓	✗	✓
	15	✓	✗	✓
Lower-bound approach	1	✗	✗	?
	5	✗	✗	✗
	10	✗	✗	✗
	15	✗	✗	✗

hold (except for C-bursts with $\xi = 5$ in the upper-bound approach). As we have seen for H3.1, there are too few data points to state valid results regarding H3+H3.1 for OpenSSL.

Since only in the upper-bound approach (and there also not for all time windows of all projects) conversation-based E-bursts do have significantly lower DTW distances with the C-bursts than message-based ones, but not in general, *we have inconclusive results regarding H3*.

3.5 What Is Discussed Within E-Bursts?

In Section 3.4, we presented the results of our quantitative analysis of the relation of C-bursts and E-bursts. The weakness of our quantitative analysis is that it does not capture whether there is actual coordination of source-code changes in temporally close-by discussions on the developer mailing list. To alleviate this threat to validity, we conducted a qualitative analysis to investigate whether our notion of coordination is reliable. We performed this qualitative analysis only for E-bursts identified via our upper-bound approach, but not for E-bursts identified via our lower-bound approach. The reason is that, in the lower-bound approach, for each developer pair, we filter the e-mails already by comparing the commit messages of temporally close-by C-bursts with the e-mail subjects. Hence, in the lower-bound approach, we only get E-bursts that are content-wise related to the C-bursts of the developer pair. As we cannot assume anything regarding the relation of C-bursts and E-bursts for the upper-bound approach, we perform this qualitative analysis to find out to which extent temporally close-by C-bursts and E-bursts are indeed content-wise related.

In a first step, we manually checked for all developer pairs with, at least, five C-bursts and five E-bursts whether the content of e-mails of an E-burst is related to the commits of the temporally close-by C-bursts. As this is a very time-consuming manual task, we only performed this for conversation-based E-bursts and only for $\xi = 15$. So, we manually looked at 56 E-bursts of BusyBox, 49 E-bursts of OpenSSL, and 766 E-bursts of QEMU. In BusyBox, 29% of these E-bursts are content-wise related to a C-burst of the same developer pair; in

Table 3.12: The results of our qualitative analysis using our mention rate, that is, the percentage of C-bursts whose artifacts (i.e., file names) are mentioned in a temporally close-by E-burst of the same developer pair.

ξ	QEMU		BusyBox		OpenSSL	
	Messages	Conversations	Messages	Conversations	Messages	Conversations
1	34%	9%	10%	5%	1%	0%
5	50%	13%	15%	8%	4%	1%
10	55%	15%	19%	10%	6%	1%
15	59%	17%	24%	12%	9%	2%

OpenSSL we found that 41% of the E-bursts are related to a C-burst, and in QEMU this holds even for 47%. Throughout our manual analysis, we identified different kinds of how C-bursts and E-bursts are related: In most cases, the e-mail subject is related to the commit message or the e-mail content even contains parts of the commit message or the patch itself. Also, certain key words are often used in commit messages or code patches that are also used in the content or subject of an e-mail. In addition to that, we also were able to match C-bursts and E-bursts by inspecting the file names of the changed files and searching for them in the e-mail subject or content. However, in cases where we decided that all the e-mails of an E-burst are not related to C-bursts of the same developer pair, we had a closer a look at the content of these E-bursts to find out what these conversations are about. It turned out that these conversations are mostly about future plans of the software project or organizational matters (e.g., coding conventions, contribution guides, workflows, or future releases). However, there are also lots of discussions regarding bugs or problems identified by users (even though we analyzed only developer mailing lists). This is also supported by the results of previous research: Guzzi et al. [GBL+13] analyzed the communication in mailing lists of OSS projects and found that only about 35% of the discussions are related to actual source-code changes. Nevertheless, in our study, up to one half of all the E-bursts are directly related to co-edits, which justifies our assumption that the developer mailing lists are used to coordinate source-code changes.

In a second step, we used an automatic approach to determine whether C-bursts and E-bursts are content-wise related. Similar to Gharehyazie and Filkov [GF17], who searched for file names of temporally close-by edited files in e-mail contents, we searched for file names of C-bursts in temporally close-by E-bursts. Using this approach, we calculated a *mention rate* for file names in E-bursts. That is, we determine the percentage of C-bursts whose artifacts (at least, one) are mentioned in temporally close-by E-bursts. In Table 3.12 we present the results of this analysis. As we can clearly see, the results differ between subject projects and abstraction levels. The mention rates are higher for message-based E-bursts than for conversation-based E-bursts. This is not surprising as the message-based E-bursts contain also single e-mails that are not related to other e-mails and, therefore, the chance for a file of a C-burst to be mentioned is higher than for conversation-based ones, due to the potentially higher number of e-mails that are contained in message-based E-bursts. Overall, we can see that the mention rate ranges from 0 to 59%, which indicates that temporally close-by C-bursts and E-bursts are related in many cases. However, the exact file names need not be mentioned when coordinating software changes. Sometimes, developers may paraphrase which file or

Table 3.13: The results of determining the percentage of commits of C-bursts whose commit messages map with the subject of, at least, one e-mail of an E-burst of the same developer pair.

ξ	QEMU		BusyBox		OpenSSL	
	Messages	Conversations	Messages	Conversations	Messages	Conversations
1	86%	31%	0%	0%	0%	0%
5	97%	43%	2%	2%	0%	0%
10	98%	44%	28%	23%	2%	0%
15	98%	45%	24%	21%	5%	0%

feature they are talking about without directly stating the name of the corresponding file. Hence, our mention rate only covers a part of the actual relation of C-bursts and E-bursts.

In a third step, we evaluated whether the e-mail filtering of our lower-bound approach is reasonable. So, we automatically checked for each commit of a C-burst whether there is, at least, one e-mail in the E-bursts of the same developer pair whose subject is equal to the beginning of the commit message. (Notice that we removed auto-generated prefixes of the e-mail subjects that match standard patterns like *Re:* or *Fwd:* or [*PATCH*] and alike before performing this analysis). We present the results of this check in Table 3.13: For QEMU, almost all commits of a C-burst are related to, at least, one e-mail of an E-burst of the same developer pair, using message-based e-mails. When only investigating e-mails belonging to the same thread, as in conversation-based E-bursts, the percentage of commits of C-bursts whose commit messages match the subject of, at least, one e-mail of an E-burst is much lower. This might be the case because we neglect single e-mails not belonging to a thread: Some of the automatically generated e-mails, whose subjects match commit messages, just contain the patch, whereas the discussion of this patch can take place in an e-mail thread different from the patch. So, the corresponding thread for the discussion of the patch can have a slightly different subject, which cannot be matched in this analysis. For BusyBox and OpenSSL, it is only rarely the case that commit messages and e-mail subjects can be mapped to each other, as these projects do not have such a strict commit policy to send patches to the mailing list as QEMU. As a consequence, our lower-bound approach (in which we also map commit messages with e-mail subjects) seems to be a reasonable filtering of the e-mails before E-burst identification, at least, for the subject project QEMU.

3.6 Discussion

In this section, we discuss our results regarding C-bursts, E-bursts, and their temporal correlation. Moreover, we address our main research question, open up perspectives, and discuss the practical value of our findings.

3.6.1 C-Bursts

Our study confirms that, for all time windows, synchronous development is not a purely random process where developers' activities are statistically independent. Knowing that our operationalization of synchronous development does not occur purely by chance, we analyze the synchronicity degree of the identified C-bursts, which is also higher than for randomly generated C-bursts. Altogether, our results show that considering the concept of synchronous development in OSS projects is well-founded. This is in line with the original study.

Code growth in synchronous development is, in most cases, lower than in non-synchronous development. This is contrary to the outcomes of the original study. The reason for this is that, in synchronous development, not only the number of added lines is higher, but also the number of deleted lines. H1.3 ignores that high coding activity in synchronous development does not necessarily incur high code growth. This also affects the implementation effort, which is, in most cases, higher in synchronous commits than in non-synchronous commits. This is in contrast to H1.4 and, therefore, also in contrast to the results of the original study. As code additions and deletions are both higher in synchronous commits, we conclude that, in synchronous development, more lines are changed than in non-synchronous development. Hence, due to higher coding activity in terms of LOC, analyzing synchronous development is useful for understanding developer collaboration.

3.6.2 E-Bursts

According to our results, the number of C-bursts and the number of E-bursts per developer pair correlate only weakly. We use the goodness of the linear fit to assess whether message-based or conversation-based E-bursts lead to identifying a stronger statistical dependence between the amount of collaboration and the amount of communication. Contrary to the original study (which only considered message-based E-bursts of the upper-bound approach), the goodness of the linear fit is lower in our analysis, but when we use conversation-based E-bursts, we get a similar linear fit as the original study reported for message-based E-bursts. The difference between our message-based results and the message-based results of the original study may be due to different sizes of the projects in terms of developers and in terms of e-mails, as our largest subject projects has 951 developers and 374 815 e-mails, whereas the largest subject project of the original study had 72 developers and 11 865 e-mails. As there is huge difference in the size of the projects, there may be also differences in the organizational structure of the projects, resulting in different numbers of E-bursts.

In the following, we compare message-based and conversation-based communication. We discuss our outcomes regarding H2 here only based on the results of QEMU, as we cannot draw reliable conclusions from BusyBox and OpenSSL. See Section 3.7 for the discussion of the corresponding threats to validity.

When we look at message-based E-bursts arising from the upper-bound approach, where we keep all e-mails for E-burst identification, we see that the number of resulting E-bursts is too high, as the goodness of fit of the linear model is rather weak. Keeping the upper-bound approach but identifying E-bursts only among e-mails that are content-wise related (conversation-based), we get a lower number of E-bursts, resulting in a better goodness of fit with the number of C-bursts than for message-based E-bursts.

When using the lower-bound approach, we recognize the opposite behavior: Conversation-based E-bursts lead to a lower goodness of fit than message-based E-bursts. An explanation might be that, in the lower-bound approach, we filter the e-mails before E-burst identification by checking whether their subject is part of the beginning of a commit message of a C-burst of the same developer pair. Due to this filtering, we already narrowed down the number of E-bursts to keep only those for which we are sure that they are related to a C-burst. When we then construct conversation-based E-bursts, the number of E-bursts is narrowed down again as e-mails need to belong to the same thread to form an E-burst. Hence, the number of E-bursts may be too small in the end.

As a lesson learned, only focusing on conversations of e-mails, of which we can be sure that they are related to C-bursts, is a too severe restriction of the view of coordination. The actual truth with respect to the identification of E-bursts is somewhere in-between our lower-bound and upper-bound approach.

When looking at the goodness of fit in general, the linear model fitted on conversation-based E-bursts in the upper-bound approach describes only less than 59% of the variance, and the linear model fitted on the message-based E-bursts in the lower-bound approach describes between 39% and 86% (depending on the time window). Such low percentages are not unexpected since developers may often send e-mails to the mailing list without contributing to the source code at temporally close time. For instance, they can comment on other issues or discuss topics independent of the source code [GBL+13]. Furthermore, multiple C-bursts may be discussed in one E-burst, whereas several E-bursts can address one single C-burst. That is, the number of C-bursts does not necessarily need to correlate with the number of E-bursts, as bursts can last differently long and discussions on interrelated topics among co-edits are possible. For the lower-bound approach, in particular, we can draw two subsequent conclusions from that: (1) Coordination of co-edits may not always happen within the same e-mail thread, as using thread information leads to a lower correlation with the number of C-bursts, and (2) coordination may not only take place via e-mails that are related to specific commits indicated by the e-mail subject and the commit message, as considering only commit-related e-mails is a strong restriction, which drastically reduces the number of E-bursts compared to the upper-bound approach.

3.6.3 Temporal Correlation Between C-Bursts and E-Bursts

Even though we found that a more nuanced view on e-mail communication (a message-based view for the lower-bound approach; a conversation-based view for the upper-bound approach) is valuable (H2), we often cannot find significant differences between the DTW distances on different abstraction levels (H3). Albeit, the empirical DTW distances are significantly smaller than the corresponding simulated ones, which is in line with the results of the original study, where the authors used correlation coefficients instead of DTW distances.

In the lower-bound approach, a lot more C-bursts than E-bursts occur per developer pair, so the number of matched C-bursts and E-bursts is comparably small. Even though there are more message-based E-bursts than conversation-based ones, the differentiation between message-based abstraction and conversation-based abstraction does not matter, since, for both abstraction levels, many C-bursts do not have a corresponding E-burst.

For the upper-bound approach, in most cases, conversation-based E-bursts lead to identifying a stronger statistical dependence between collaboration and communication than message-based E-bursts. Thus, if there are enough communication data, a more nuanced view on the communication of developers describes coordination among developers more precisely than simply looking at single messages.

Nevertheless, the temporal correlation between C-bursts and E-bursts is for both the upper-bound and the lower-bound approach higher than in the null model. This demonstrates a significant statistical dependence between collaboration and communication, which implies that developer coordination is actually taking place, but depends on many variables.

3.6.4 Research Question and Perspectives

In our main research question, we looked for the best abstraction level of developer coordination in terms of a stronger statistical dependence between the technical activities and the social activities on the mailing list. Our results suggest that lifting the study of developer coordination to a conversation-based model is worthwhile and sometimes even required. However, for the temporal alignment of coding and communication, many different aspects matter, which shall be explored in further studies.

To summarize, the correlation of collaboration with communication activities depends on many variables. We did not observe any universal developer behavior when comparing the coordination using message-based and conversation-based E-bursts. To some extent, the relationship between collaboration and communication is project dependent, which needs to be taken into account to refine the general measurement method toward a specific setting.

As the null model essentially captures the case in which developers' technical activities and social activities are performed independently at random and, since we see a departure from the null model in our empirical data, there is a stochastic dependence between developers' technical activities and their social activities. This is in line with the outcomes of the original study and indicates some extent of alignment in terms of time and structure between the technical realm and the social realm, which is sometimes referred to as socio-technical congruence [CH13; CHC08; CWH+06; MJT+22]. Our approach relies on a more dynamic interpretation of socio-technical congruence than in most previous work: We use dynamic time warping to identify the alignment of technical and social activities, whereas previous work adequately ignored the dynamic nature of the phenomenon and searched for an alignment within static time windows, as, for instance, in the work of Joblin et al. [JAM17].

The stochastic dependence between collaboration and coordination that we have identified has practical value because it helps to reduce the uncertainty when making predictions. For example, if we know that a pair of developers was involved in a huge number of C-bursts and we know that C-bursts and E-bursts are dependent, we may build a model that predicts for their joint technical activity the need for coordinating their work. In addition, we could think about exploiting the relationship between two developers represented by a C-burst to predict which files a developer is likely to work on next by considering past C-bursts. We could also search for missing dependences between collaboration and coordination and investigate how the quality of the development process and the developed artifacts are affected. For instance, one could check whether there are more bugs and other issues on a specific artifact if there is no E-burst related to a C-burst on the considered artifact, to get an even more detailed view on how software development is influenced by coordinating activities. Finally, as we have identified stronger statistical dependences when using higher-level views on the abstraction level of coordination (i.e., a more content-related view), it is more feasible to execute the described ideas using higher-level views.

3.7 Threats to Validity

As always in empirical studies, the validity of the results of our study may be threatened by different factors. As suggested by Wohlin et al. [WRH+12], we have grouped the potential threats into the four categories that were introduced by Cook and Campbell [CC79].

Internal Validity In our study, we do not consider changes to header files, documentation files, and build files. This affects our results only barely, though, as build files or documentation files are changed comparatively rarely. As the studied projects predate the existence of the version-control system Git, the commits in the Git history at the beginning of the analyzed time range of all three subject projects had originally been imported from the previously used version-control system SVN, which had a slightly different operationalization of commits. However, to the best of our knowledge, this does not threaten our notion of C-bursts, as the import of SVN commits into Git did preserve authoring timestamp, author, and code changes.

Construct Validity Our results depend on the algorithms that we use to identify C-bursts and E-bursts, and also on the techniques that we use to measure their synchronicity. To quantify the significance of C-bursts, we designed a metric that considers the size of commits and allows to quantify the magnitude of the overlap among commits of a C-burst. This way, we were able to demonstrate that C-bursts are not artifacts of a purely random process. To measure the synchronicity of C-bursts and E-bursts, we rely on the well-established DTW technique (see Section 3.1.3), which can be properly restricted regarding the distance measurement (i.e., the Sakoe-Chiba band [SC78]). By comparing our empirical observations to null models, we showed that the observed curves convey information that is significantly different from purely random information, which justifies the choice of our measuring techniques.

We rely on mailing lists as the only communication channel, although developers may happen to use further channels (e.g., personal e-mails or verbal communication) [SSF+17]. We mitigate this threat by selecting only projects that have a historically rich and well-established mailing list for discussing code changes and which have mandates regarding patch submission to the mailing list prior to being accepted [DP03; RGS08; RLM19; Som10]. In addition, more recently introduced social coding platforms (e.g., GitHub) are too young for history analysis, whereas the mailing lists of our subject projects date back more than 10 years (see Table 3.2). Also, recent research has shown that a substantial number of OSS projects still uses mailing lists as primary channel for developer communication [MAJ+20; RLM19; SSF+17; YZF23].

Conclusion Validity For both commit and e-mail time series, we generated 100 simulated time series each per project. Due to the sheer size and complexity (i.e., generating an individual simulation of the e-mail time series separately for each developer pair), we were not able to achieve this for the e-mail time series of QEMU (which has 451 725 developer pairs). While this threatens the validity of the results of H3.1, all other results confirm that the involved processes are significantly different from the null model and, thus, not purely random.

The sparseness of the data threatens the validity of our study: The number of developer pairs having, at least, one C-burst and, at least, one E-burst is low ($\sim 0.2\%$). Yet, this is expected since not all developers actually collaborate. Particularly, BusyBox and OpenSSL have only few developer pairs involved in, at least, one C-burst and one E-burst. So, we cannot draw reliable conclusions from these projects regarding the relationship between the number of C-bursts and the number of E-bursts. Nonetheless, as the number of E-bursts is narrowed down to zero with the lower-bound approach in some cases of OpenSSL, we should take a closer look at the different code-contribution practices of the different projects: Whereas there is a policy in QEMU[4] to send patches to the mailing list and discuss them there, other projects, such as OpenSSL, do not have such a strict code-contribution policy. As a consequence, on the QEMU mailing list, there are lots of e-mails that contain a patch and, therefore, automatically contain the title of the corresponding commit message in their e-mail subjects, whereas in other projects like OpenSSL this is not the case that often. Hence, our lower-bound approach is limited to the strictness of the patch-contribution policy of the respective subject project.

Finally, we did not perform a linguistic analysis of the e-mail data. Hence, the communication on the mailing list may partly concern other issues than coordinating source-code changes. We alleviate this by considering only e-mails of developers who also contributed to the source code of the project. Moreover, we performed a qualitative analysis to check whether there is a content-wise correlation between temporally close-by C-bursts and E-bursts (see Section 3.5).

External Validity We have analyzed three different OSS projects, which differ in size, commit policies, and application domain. Due to the high computation time and huge memory consumption of our approach, we cannot analyze more than these three subject projects within a reasonable amount of time and memory. While one cannot generalize our findings arbitrarily—as always in such a study—we have substantial data for three large, highly active, and widely deployed OSS projects, which gives us relevant insights into the behavior of collaboration and coordination at different abstractions levels of communication.

3.8 Conclusion

The success of software projects, in particular, large-scale, globally distributed projects, relies essentially on the coordination of co-edits to the source code, as previous work has shown [CH05; CH13; KSD11]. Co-editing source code is a common way of performing bug fixes, refactorings, enhancements, and adding new features concurrently [Sin10]. We investigated the relationship between co-editing activities and communication on the mailing list for three highly active and widely deployed OSS projects using different abstraction levels.

We found that a more nuanced view of communication substantially increases the correlation between co-editing and coordinating e-mail activities compared to a simple message-based view, which is reasonable since coordination of developers comprises collections of conceptually related e-mails. We did not observe a general picture regarding the temporal correlation between co-editing source code and e-mail communication, though, which depends to a good extent on the project setting at hand.

Overall, we found evidence that a more abstract and higher-level perspective captures developers' coordination activities more accurately than a rather low-level and sole message-based perspective. This is not unexpected because developers typically think in terms of topics when building mental models of software rather than in terms of individual text messages. Further studies in this area should take this perspective into account.

4 Group Dynamics and Group Stability in Open-Source Software Projects

This chapter shares material with Bock et al. [BSA22].

Open-source software (OSS) projects often depend on a relatively small group of developers who are accountable for a large share of code contributions and coordination efforts. These *core developers* are supported by a large number of *peripheral developers*, who invest less time and effort individually, but together add a substantial part of the value in OSS projects [CS17; JAH+17; SRS+12]. Peripheral developers often pursue a particular aim, such as fixing a particular bug or getting a feature they need [KJR+16]. This way, core and peripheral developers form *communities*, which make significant contributions. Within the communities, shifting priorities and tasks leads to the formation of temporary *sub-groups*, working on a topic or task for a limited time only [Bir11].

Social sciences and software-engineering researchers, as well as practitioners who strive for the optimal software development process, seek to understand and predict the social dynamics of OSS projects. Metrics that operationalize social dynamics correlate with the emergence of code deficiencies and bugs [CTB+15; WCR+14] and help predict project success [CB16]. For example, a high stability of a project's organizational structure can result in several beneficial characteristics, including robustness and scalability [DM03]; tensions between single developers can cause significant distortions in group structures and substantial developer turnover [JS05].

Network analysis can help to investigate structural distortions and assess the stability of a project's organizational structure: Innes et al. [IRP+17] used network analysis of communication data to understand the genesis of conflicts in social networks. In general, network and graph mining methods can help understand social phenomena in OSS development [BGD+06; Duc05; JMA+15]. In this chapter, we devise a measurement and analysis method that relies on graph mining methods to explore two facets of the social coding process: the strength and stability of sub-groups, and the relationship between communication and collaboration (i.e., co-editing source code). Methodologically, we map a *multiplex* network structure to a four-dimensional data structure (i.e., a tensor). Multiplex networks consist of several networks that share the same set of vertices but differ in types of interaction [IB16]. A multiplex network takes several modes of interaction simultaneously into account without mixing them up—such as communication and co-editing source code in our case. We apply a *tensor decomposition* to detect temporary sub-groups in the networks. We visualize the result of the decomposition to simplify the detection of groups and anomalies. Finally, we use a trend extrapolation to investigate whether we can leverage the sub-group dynamics to predict future interaction events. With this last step, we also validate whether our assumed overlapping group structure describes the OSS group structure well. Network analysis is a powerful tool for studying organizational structures in a non-intrusive way, without changing or influencing the observed organizational structure or system [KC18]. We seek to contribute a methodology to investigate the effects of social structures and their emergence in more detail, to support research on organizational stability like research on the re-emergence of sub-groups in self-organizing OSS communities after major disruptions [WS21].

To demonstrate the practicability of our method, we use mailing-list data and file-editing data from 10 substantial, popular OSS projects of varying size. Ultimately, we seek to lay technical and methodological foundations for research on social dynamics in OSS development, which can complement other approaches to social-network analysis in OSS research [JPW17].

Our results show different levels of stability of social relationships across different OSS projects. We found significant improvements in predictive accuracy when concentrating on the central sub-groups and reducing the weight of peripheral developers—which means that stable sub-groups arise that last for a couple of years in all projects that we investigated. In some of the projects, we also saw high levels of congruence between the groups that discussed via the mailing list and the groups that were collaborating on the source code. When we augment the target number of groups that the decomposition is supposed to identify, more detailed insights into role differentiation are possible. Our method is capable of identifying single developers who complement *either* the programming groups *or* the discussion groups only. In some projects, we identified a stable group of people that take over communication and programming activities right from the beginning, and that persisted in later phases and took over the coordination activity. Moreover, we find that growth in the number of developers and co-editing events is usually not complemented by communication activity growth. Our results imply that, even though all of our subject projects are successful and have a vivid contribution history, they differ structurally in the strength of social relationships and the coherence of communication and collaboration. This can be related to different ways of using mailing lists. We found stable social structures in all projects, and a consistent improvement of link prediction performance when concentrating on the particular sub-groups instead of all developers. However, against common belief, our results indicate that, in many projects, the sub-group structure that arises from developer communication does *not* align with the sub-group structure that arises from co-editing source code, indicating that the congruence between communication activities and co-editing activities is not always pronounced.

In summary, we make contributions in four areas:

1. From a methodological perspective, we advance the use of multiplex network modeling of the organizational dynamics of software projects: We devise a modular, multi-step method to detect developer groups using established community-detection algorithms and predict their behavior using widely-used forecast algorithms. In particular, we use a canonical tensor decomposition and combine it with a state-space trend extrapolation to detect temporary sub-groups and predict future developer interactions. With this exploratory approach, we lay the foundations for further inductive research for finding patterns in social networks.

2. From a software-engineering perspective, we provide insights into the strength and stability of group structures in OSS development, as well as into the diversification of roles and the congruence of communication and collaboration behavior.
3. From a practitioner perspective, we propose a method to track important changes in a project's organization, identify sources of problems and change, and thereby recognize instabilities and irregularities in project and process management. Such irregularities or changes in the group structure of OSS projects could arise when, for example, core developers leave or join a project, or when they pursue other tasks than they did before.
4. From a research perspective, we provide a measurement and modeling framework in form of a replication package (consisting of analysis scripts as well as links on how to gather the raw data used for our study) on our supplementary website: `https://se-sic.github.io/paper-groupdynamics-oss-tensordecomposition/` and `https://zenodo.org/records/10556710` .

4.1 Background & Related Work

In this section, we first provide an overview of related work that investigated the stability of organizational structures in OSS projects and explain the research gap we seek to fill. Thereafter, we provide background information on the statistical tools and concepts that we use for our method to detect group structures in developer networks and to validate their stability.

4.1.1 Investigations of Organizational Stability in the Literature

Researchers in management science early started to investigate the emergence of governance, communities, and organizational stability in OSS projects [OF07; OJ07]. Previous work in software-engineering research has shown that the organizational structure of a software project, in particular the coordination among software developers, has an influence on software quality [e.g., CH13; GHP99; HMR06; KS95; KSD11; NMB08; TLV13; ZCM+17]. However, Mauerer et al. [MJT+22] have recently shown that socio-technical congruence is a "complex and multi-faceted phenomenon", that varies across projects and their evolution. In general, there is a growing corpus of research on coordination and collaboration of developers in OSS projects [CS17; RCO19; STG19].

Network representations are core to the analysis of OSS projects: Developers constantly create connections when communicating or changing code. From a graph or network perspective, this means that developers are represented as vertices, and interactions between developers as edges (which are also called *links*). Social and socio-technical network analysis helps analyze the determinants of software quality [MW11], track the evolution of OSS project organization [JAM17], for learning about code deficiency [CTB+15; WCR+14], and understanding OSS success [GLM06; TKL+15]. As many disciplines in organization science seek to learn from the self-organizing nature of OSS projects, there is a substantial corpus of work that investigates network structures to explore developer characteristics and organizational structures in OSS projects [e.g., CLL+17; HKC+11; HSA20; JAH+17; JSS11; LRG+06; PD08; SSS17; TMB10; TZF20; ZYW11]. For example, Palomba and Tamburri [PT21], and Tamburri et al. [TPS+19] investigated network characteristics to discover community patterns and to predict community smells (e.g., developer groups that do not communicate with each other or developers who dominate discussions regarding almost every topic). They found that social-network characteristics, in particular socio-technical metrics, are most valuable for predicting community smells in OSS projects. We will use a network approach to investigate the dynamic stability of social ties and sub-groups of OSS projects. For this purpose, we look at both communication (social) and co-editing (socio-technical) among developers, modeling developers as vertices and interactions (communication and co-editing) as edges. Co-editing is, in our context, the joint creation or joint edits of a software artifact (e.g., file).

With this operationalization, Joblin et al. [JAM17] investigated the collaborative dynamics of 18 large OSS projects. They found that, over time, "the organizational structure of large projects is constrained to evolve toward a state that balances the costs and benefits of developer coordination". As one of the typical long-term patterns, Joblin et al. found that *modular* group structures tend to emerge in developer collaboration [JAM17]. The "mirroring hypothesis" suggests that the social structure of a software project mirrors its code structure because of the coordination needs that software modules create [CB16]. This means that group structures among developers may reflect modular structures in code. For example, there are several sub-modules in the Linux kernel. Each of these sub-modules involves different files and developers, but single developers can also contribute to more than one sub-module. Basing on the theory of mirroring, we assume that groups are generally overlapping, and one voluntary developer can be part of several groups.

Previous findings on the strength of social structures in OSS projects are contradictory. One stream of research sees relatively weak social structures, and a high degree of flexibility. Bird et al. [BPD+08] found that developer networks of OSS projects show a high degree of volatility and that communication tends to adapt *ad hoc* to current tasks in programming instead of forming a stable organizational structure. Bird et al. [BGD+06; BPD+08] and Shihab et al. [SBA+10] did not find any structure beyond the ad-hoc collaboration on current tasks, which led them to the conclusion that there is little long-term stable organizational structure in OSS projects. In contrast, Howison et al. [HIC06] showed that, in OSS projects, the degree of group strength and stability also depends on whether developers are core or peripheral members: Independently of project size, the core groups are more stable than peripheral, temporary coalitions. In combination with the large share of peripheral developers in OSS projects, this may explain why measures that do not differentiate between core and peripheral developers fail to identify stable social relationships. Both the presence and absence of long-term collaboration foster a strong congruence of communication and collaboration tasks. Previous research found that collaboration creates communication needs, as it requires the exchange of information on the source code and coordination of programming tasks [CH13; CWH+06].

Mauerer et al. [MJT+22] explored whether and how socio-technical congruence affects software-quality metrics. In particular, they investigated whether the number of bugs or code churn (the number of changed lines) are statistically related to socio-technical motif congruence (which is a notion of socio-technical congruence that incorporates that developers who work on the same file should communicate with each other). Their findings reveal that the degree of socio-technical congruence is not related to the number of bugs, nor to code churn.

Ashraf et al. [AMM+21] investigated whether developer groups derived from socio-technical issue networks of OSS projects are related to sets of developers contributing code to the same subsystem of an OSS project. Their results indicate that the developer groups arising in communication, in general, do not overlap with the developer groups derived from jointly editing source code in the same subsystem. Furthermore, they identified that developer groups in communication are rather task-driven and unstable over time and, thus, communicate with varying developers that prematurely work on other subsystems, whereas developer groups derived from the joint contribution to the source code of a subsystem are rather stable.

The multi-dimensionality of group structures is also related to a role differentiation that may arise in social organizations. Depending on tenure, experience, and personality, developers may prefer to contribute only to the collaboration or the communication mode of the network or to do both simultaneously. If developers can freely choose their focus, they can also be active in whatever way they want. They may prefer programming, coordinating, or merely discussing ideas and technical solutions or organizational issues without actually contributing code [LC03]. Consequently, it is advantageous to look at different forms of interaction simultaneously to understand the social structure of a project. Tymchuk et al. [TML14] showed that relying on a single data source for measuring collaboration can be problematic. This simplification can lead to false conclusions on group structures, the strength of relationships, and the importance of single developers. Gandhi et al. [GKD+19] showed that multi-layer modeling of different types of relationships of developers adds value to the analysis of OSS developer interaction by revealing relationships that would be undiscovered in the analysis of uni-dimensional links among developers.

Previous research lets us expect a partial congruence of communication and co-editing. Looking at the content of communication, researchers have found that indeed a large share of communication in OSS projects is centered on ad-hoc needs of programming tasks. Guzzi et al. [GBL+13] conducted a text analysis of mailing-list communication to analyze the interplay of co-editing and communication in OSS projects at a content level. Their insights suggest that an important part of communication efforts goes into activities that are *not* directly related to programming tasks. In particular, Guzzi et al. found that mailing lists are an important, though not the only, medium of developer communication. They categorized the topics of communication and found that only 16% of all threads were treating technical infrastructure. Communication about project status (e.g., planning and communication about releases and due dates) is accountable for around 7% of all mailing-list communication. This part of the communication is independent of direct implementation issues. Moreover, social interaction (i.e., threads about topics such as social norms, contributors, acknowledgment, and coordination) is accountable for about 6% of all threads. So, a significant share of communication is not on implementation issues and reflects social relationships that go beyond the urgent technical needs of collaboration and programming [GBL+13]. Mannan et al. [MAJ+20] showed that roughly 89% of the technical discussions of an OSS project appear to take place on the mailing list of the project. Not only the core developers, but also peripheral developers participate in these implementation-related discussions, which is an indicator that various kinds of developers having different roles in the project are involved in such discussions. Thus, mailing lists are an important and comprehensive communication channel when investigating the relationship between communication and programming activities among developers.

4.1.2 Tensor Decomposition, Spectral Stability, and State-Space Extrapolation

For our method to detect group structures in OSS projects and to validate their stability, we rely on multiple established statistical concepts, which we briefly introduce in the following.

Tensor To obtain a multiplex network view on OSS projects, in which we can investigate several interaction channels simultaneously without mixing the different channels up, we use the concept of a tensor. A tensor is a higher-order array: An array of order zero is a scalar, an array of order one is a vector, an array of order two is a matrix, and a higher-order array is a tensor [KB09; WD16]. We use the tensor as a four-dimensional data structure to model the edges between developers on different interaction channels over time.

Latent Factor Model *Latent factors* (also called *latent variables*) are unobserved variables that are "hypothetical constructs" which have an effect on observable variables [Eve13]. *Latent factor models* are used to describe the dependencies of observable variables by a small number of latent factors [Eve13; LX23; Thu31]. Thus, latent factors usually are displayed as a linear combination of observable variables [LX23].

Canonical Tensor Decomposition Tensor decomposition is a latent factor model that is a well-established and fast method to detect group structures when multiple networks are assumed to have a common underlying group structure (as, for instance, in communication and programming activities of developers); it is robust to various statistical data distributions [CCZ+19]. *Canonical tensor decomposition*, also known as *CANDECOMP/PARAFAC* or *CP* decomposition [CC70; Har70; Hit27; KB09; Kie00], is "one of the most popular tensor decomposition methods" [FFG18] and has the goal to represent a tensor "as the sum of a finite number of rank-one tensors" [KB09]. In particular, tensor decomposition plays a vital role in knowledge discovery in multi-dimensional networks where computational complexity quickly grows, and matrix-based methods miss the dependency of different network modes [FFG18]. As Gauvin et al. [GPC14] pointed out, a matrix-based approach would require the aggregation of all data in time, which leads to the loss of temporal information. Alternatively, in a matrix-based approach, several interaction channels could be aggregated to keep temporal information, but then there is a loss of interaction-channel information. A tensor decomposition avoids this loss of information. For that reason, we use tensor decomposition to map developer-interaction data to a higher-dimensional tensor structure and decompose it into time, person, and channel-related patterns with a higher-order tensor decomposition [SCA+12]. The tensor-based approach has the advantage of keeping time ranges and interaction channels separate. The perception that overlapping sub-groups in developer networks arise or vanish can be translated into a *dynamic low-rank factor model* [AGH+14a; Hof09] that is based on the assumption that collaboration and communication activities are driven by unobserved "tasks" or groups, which lead to developer clustering, and cross-channel and dynamic dependency.

Rank Reduction The combination of a tensor decomposition with a selection of a subset of the available factors is called *rank reduction* or *low-rank latent factor model* [HRH02; Hof07; Hof09; Hof11]. By restricting the number of factors, the data is projected into a lower-dimensional space, making the data more concise and reducing noise [SH05]. Thus, a reduced-rank tensor decomposition makes sure that the entries of the tensor "are expressed as products of low-dimensional latent factors" [Hof11]. The low-rank factor model has been used in the field of data mining for link prediction [AFH+12; AGH+14b; Hof09; KBV09; SCA+12], as well as for community detection [AGH+14b].

Canonical Tensor Decomposition vs. Principal Component Analysis As we expect many readers to be familiar with the factorization of matrices via principal components, we explain the similarities and differences between canonical tensor decomposition and principal component analysis via spectral matrix decomposition. A principal component analysis is a method to infer two-dimensional latent factors from a matrix [Jol02]—like a canonical decomposition is a method to infer higher-dimensional latent factors from a tensor. The low-rank tensor decomposition is similar to a principal component analysis in that it *projects* or *maps* n-dimensional data into a lower-dimensional coordinate system spanned by rank-one *components*, which is one way to infer *latent factors*. Like in matrix decomposition, the number of possibly extracted components or factors is equal to the *rank* of the matrix or tensor, respectively. There are, however, two main differences in the choice of the number of components to extract. First, the computation of the rank of a tensor is NP-hard [SH05]. Second, the spectral decomposition of a matrix always produces the same first factors (ordered by importance), no matter how many latent factors are being extracted [Jol02]. The latent factors in tensor decomposition, however, depend on the total specified number of factors. When the specified number of factors (which we call R in the remainder of this chapter) is chosen too small, interesting patterns will be overlaid. When R is chosen too big, then spurious patterns arise, which have nothing to do with real-world group dynamics. In the case of tensor decomposition, "repeatedly subtracting the dominant rank-1 tensor is not a converging process" [SH05, p.794]. Therefore, the number of components influences the results, unlike in principal component analysis, and the choice of R is non-trivial and is prone to overfitting. For repeated estimation, the tensor decomposition can find different solutions for the same R, as its results depend on the random initiation. Therefore, it is important to set a seed (when doing the computations). To summarize, the tensor decomposition is an exploratory instrument to find interesting patterns, which should then not be over-interpreted (especially for large R).

Spectral Growth & Stability When extracting latent factors via tensor decomposition from a tensor that consists of a time dimension, it can happen that the importance of a latent factor (i.e., in our case, the importance of the corresponding group structure) can be subject to dynamic change. The assumption of constant factorial patterns with trends in dynamic importance has been discussed by Kunegis et al. [KFB10] under the name of "spectral stability": They developed a *spectral evolution model* and proposed to infer the *effects* (i.e., latent factors) that affect the vertices in the network from two consecutive time slices of a network (via matrix decomposition), and to extrapolate the changes in the dynamic weights to predict future links. The designation "spectral" results from the fact that their argumentation bases on spectral matrix decomposition as a factor extraction method. Kunegis et al. find that, the more stable the interaction between the vertices, the better an extrapolation of the temporal importance to the future performs for link prediction.

To evaluate their spectral evolution model, they compared the performance of their method to more restrictive growth models. Such restrictive growth models include the assumption of constant relative importance of the single unobserved components such as triangle-closing kernels and path-length models [LBK+08]. Kunegis et al. found that their method performs better in link prediction if the relative performance of the single underlying growth-driving patterns (whatever these patterns are) changes over time. They attribute the gains in predictive performance to the additional freedom that there is no parametric growth curve imposed on the temporary weights of the decomposition [KFB10]. As we base our methodology on tensor decomposition, we can profit from more information and adapt their extrapolation strategy in several regards.

State-Space Time-Series Prediction Models State-space models are statistical tools to analyze time series of data, which are widely-used to estimate parameters or perform forecasting [DK12]. In this regard, a time series is a set of observations that is temporally ordered [DK12]. In particular, state-space models assume that unobserved variables determine the evolution of a system (i.e., its state) over time. These unobserved variables are assumed to be related to observable variables, and the state-space model specifies the relationship between the unobserved and the observed variables [DK12]. This relationship is usually characterized by a *trend*, which is "a slowly varying component", and by a general *level* of the relationship [DK12]. In this context, estimating the unobserved variables based on the total set of observations is called *smoothing*. However, a multitude of different smoothing functions can be used for this process, which makes state-space models flexible for different application scenarios. For further details on state-space models, especially from a more formal and statistical point of view, we refer to the literature [e.g., Aok13; CK07; DK12].

Simply said, the state-space model is a time-series model that estimates that current information is more valuable than old information for the prediction of a time series. This estimation is in line with our assumptions on developer collaboration, as knowing about the current collaboration of developers might be more valuable for predicting future collaboration than knowing that certain developers had collaborated several years ago. To avoid overfitting and vulnerability to a small forecast horizon, we use additive double exponential smoothing, which is a state-space extrapolation method proposed by Hyndman et al. [HKS+02]. For details on additive, multiplicative, and other models, see Hyndman et al. [HKS+02].

4.2 Research Questions, Method, and Models

In this section, we translate our assumptions on the nature of group structures in OSS developer networks into statistical models. For this purpose, we first formulate three research questions and then explain our method and the models that we use to answer them.

4.2.1 Research Questions

In our study, we aim at answering three research questions regarding group structures in OSS projects and their stability. We encode different facets of developer interactions into different models and use statistical tools to model these facets. The facets we model include repeated interaction, stable social groups, and dynamic development of groups. The core modules of our method consist of a clustering of the developers, and a time-series approach to predict the future level of activity of the identified developer groups. We use tensor decomposition and a state-space time-series prediction model to implement these modules. While the individual statistical tools we use are well established, their combination to validate assumptions on developer behavior is novel. Our research questions consequently include both the capability of our method to model developer behavior and the insights into the developer behavior itself.

First of all, we are interested in whether stable group structures do emerge in OSS projects, leading us to our first research question:

RQ$_1$: Are there stable group structures in OSS projects? That is, are there groups of developers that steadily interact with each other during the project's evolution? Or are there no stable group structures, merely developers who just rally round certain tasks and vanish afterwards?

As we model different interaction channels simultaneously, namely co-editing source code and communication, there might be commonalities or differences of the group structures of the different channels. According to the "mirroring hypothesis" [CB16], there shall emerge a state of socio-technical congruence between organizational structures (e.g., communication among developers) and structures arising from programming activities, which results in joint group structures. This brings us to our next research question:

RQ$_2$: Does the communication behavior of developers result in the same group structures as arises from co-editing behavior? To what extent do the group structures that emerge from communication and from co-editing source code overlap in terms of developers who participate?

To identify group structures and to obtain insights into their emergence and evolution, we devise a method combining different statistical tools. We suggest four models addressing different facets of the interaction of developers in OSS projects (such as past interactions of developers, which might still be present in the future due to acquaintance and common working topics). In addition, we propose two baseline models for comparison. To answer our research questions, we compare the models' predictive performance on 10 subject projects, covering different project sizes and project domains, to demonstrate how well the different models perform in different situations. The predictive performance of the different models helps us to judge the stability and importance of the detected group structures: If a certain facet of developer interaction is present, the corresponding models should have a better predictive performance compared to the models which do not consider this facet. As we use predictive performance to evaluate our models, we aim at answering the following research question to get a better understanding of the stability and endurance of interactions among developers:

RQ$_3$: Does considering past activity in co-editing or communication improve the prediction of future co-editing or communication? Following the "mirroring hypothesis", can the prediction on one channel be improved by incorporating past activity on the other channel respectively?

To explore the capability of our method to model developer behavior, we use descriptive insights and decomposition insights from the application of our method to our subject projects. Our research questions on the insights into developer behavior will mainly be answered by analyzing the predictive performance of our models, as the predictive performance helps us to figure out whether a certain facet of developer interaction is present in OSS projects.

4.2.2 Overview of the Method

We devise a methodology to study sub-group stability. Previous studies, such as the studies by Joblin et al. [JAM17], Bird et al. [BPD+08], and Shihab et al. [SBA+10], considered a single mode of interaction, such as e-mail communication, for finding social coalitions. In contrast, we consider dynamic group structures in the joint communication and co-editing behavior of OSS developers. We use a model that assigns a probability of interacting for every pair of developers in a specific way at a specific time. This probability depends on whether the two developers are part of a sub-group, and whether this sub-group is currently active or not. We define a group to be a temporal coalition among developers that manifests in communication and co-editing activity, where groups are non-exclusive, and every developer can be "fractionally" involved in an unlimited number of groups. By *co-editing*, we mean *collaborative source-code creation or editing* within a certain time window, which can be tracked and assessed via version-control data, as introduced by Joblin et al. [JMA+15] as a notion of collaboration. So, co-editing captures all the edits developers make to a commonly changed source-code artifact (i.e., file) within a certain period of time. We expect that co-editing will manifest, at least, to a certain degree in the developers' communication. That is, we expect a positive correlation between co-editing and communication events. We operationalize communication by tracking the exchange of information via mailing lists. In our operationalization, we speak of *communication* when two developers contribute to the same mailing-list thread within a certain period. Unlike Gandhi et al. [GKD+19], we emphasize a *dynamic* view of these multi-layer structures, and discretize the network into multiple time ranges of 3 months duration and track changes over time.

We base our models and analysis on the following assumptions, as illustrated in Figure 4.1: In some projects, alliances may arise *ad hoc* depending on spontaneous coordination requirements, whereas in others, strong social bonds may prevail, leading to more stable long-term group structures. We assume that there are overlapping groups in OSS projects, that manifest in co-editing and communication behavior. The more regularly developers contribute, the more critical their role for the single group. Given observed communication and co-editing activities, we can infer the groups even if we do not have explicit information on group structures. When designing our analysis, we seek to cover three main facets discussed above: First, if group structures are more stable among core developers than among peripheral developers [HIC06], an algorithm that seeks to identify stable group structures can benefit from concentrating on the core group of developers and down-weighting the importance of peripheral developers. Second, we want to assess the stability of coalitions over time, instead of describing a group structure descriptively. Third, we want to consider multiple channels of developer interaction simultaneously.

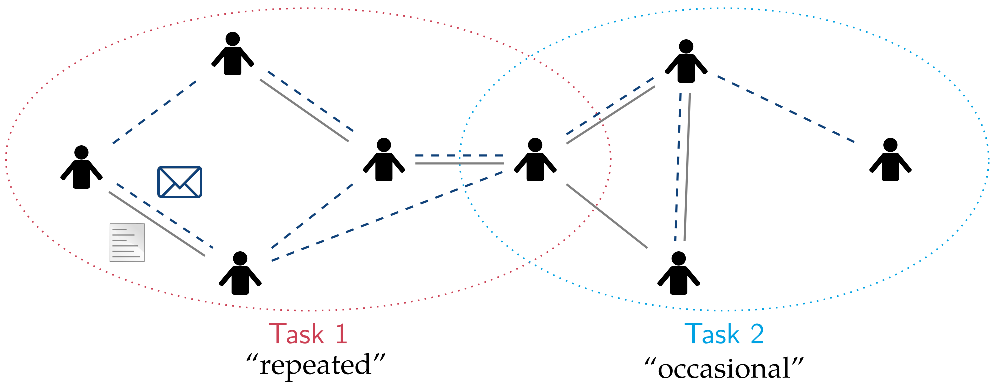


Figure 4.1: Graphical representation of our assumptions: Unobserved "tasks" drive co-editing (solid lines) and communication (dashed lines), causing overlapping group behavior. Our definition of groups involves (a) a subset of developers, (b) the usage of the interaction channels communication and co-editing, (c) within a limited period. Every developer can be part of multiple groups with varying intensity.

4.2.3 Network Representation of Open-Source Software Development

Group structures describe the state of a network *for a certain period* and shed a topological perspective on the network. To operationalize these periods, we choose to use subsequent time windows (to avoid that a specific developer interaction appears in more than one time range), as described in Section 2.3.2. Like Zimmermann et al. [ZZW+05] and Gall et al. [GHJ98], we call the act of two edits to the same source-code artifact (i.e., file) by two developers within a specific time window a *cochange* event. Further, we call the act of mailing-list communication between developer i and developer j a *mail* event if i and j both sent an e-mail to the same thread on the mailing list within some time window. Many OSS projects rely on mailing lists for communication and the discussion of new patches and code [MAJ+20; RLM19; SSF+17]. In some projects, every patch must be sent to a mailing list for general discussion before the developers' proposed changes are incorporated into the software. In the project policy of QEMU[1], for example, the process of submitting a patch to the mailing list is strictly regulated. The need to communicate to get changes rolled out leads to a close relationship between development and mailing-list communication [BPD+08; XF14]. The mail network does not cover all facets of communication, though. It is possible that data from other communication channels, such as issue data from GitHub or other version-control platforms, could bring additional information about communication. In the ideal case, we would include all available means of communication, which would provide the most detailed insights into the interplay of communication and co-editing. However, mailing lists have high coverage, and researchers showed that they play a non-negligible role in OSS development [MAJ+20; SSF+17], and a dominant role in our subject projects (see Section 4.3.1).

We denote a *cochange* event between developers i and j in time range t by a scalar binary value $z_{ijCt} \in \mathbb{B}$. If developers i and j have co-edited (at least) one source-code artifact in t, then let $z_{ijCt} = 1$, and $z_{ijCt} = 0$ otherwise.[2] If developers i and j have contributed to (at least)

1 `https://wiki.qemu.org/Contribute/SubmitAPatch/` (accessed at 2019-02-15)

2 In this chapter, we denote scalars by lowercase italic letters, vectors by bold lowercase letters, matrices by bold capital letters, and higher-order arrays by underlined bold capital letters.

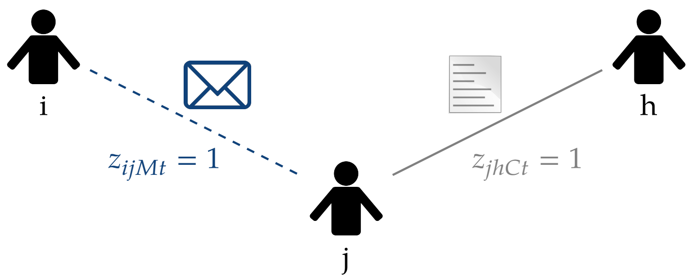


Figure 4.2: Our central data structure: Two developers are connected at time t if they have either communicated ($k = M$, *mail*, dashed line) or co-edited ($k = C$, *cochange*, solid line).

one common e-mail thread within t, let $z_{ijMt} = 1$, and $z_{ijMt} = 0$ otherwise. More generally, z_{ijkt} is an interaction event, where $k = C$ stands for *cochange* and $k = M$ stands for *mail* events. Much like previous work, we do not distinguish who wrote the message first or contributed first, therefore, $z_{ijMt} = z_{jiMt}$ and $z_{ijCt} = z_{jiCt}$. The event z_{iikt} is not defined, since a vertex cannot have a relationship with itself. For the tensor decomposition, these values are set to 0. We illustrate our notation in Figure 4.2 for better comprehensibility.

The binary *cochange* and *mail* events $z_{ijkt} \in \mathbb{B}$ can be arranged in a fourth-order binary tensor $\underline{\mathbf{Z}} \in \mathbb{B}^{N \times N \times K \times T}$, where N is the number of developers, K is the number of interaction channels (here: *mail* and *cochange*, $K = 2$), and T is the number of time ranges. When $z_{ijkt} = z_{jikt}$ (as in our case, because we do not distinguish senders and receivers of e-mails), it is said that the tensor is *symmetric in the first and second mode*. $\mathbf{Z}_{..kt} \in \mathbb{B}^{N \times N}$ is a two-dimensional tensor slice—a matrix that contains all interactions at time t in channel k. As z_{iikt} has been set to 0, the main diagonal of every tensor slice $\mathbf{Z}_{..kt}$ contains only zeros.

4.2.4 Predicting Interaction

To approximate communication and co-editing structures in OSS networks, we mine interaction behavior of developers. The observed data allow us to infer the *probability* of interaction among developers in several interaction channels for future interactions. For inference, we combine the observed data with a model, and the *fit* and *predictive performance* of this combination provide information on the correctness of our assumptions on the behavior of developers. So, the main goal of our model is not prediction of future states, but the validation of our assumption on developer interaction behavior. Our basic model is the *overlapping group structure with dynamic importance*, as illustrated in Figure 4.1: Developers work together in overlapping groups or sub-groups. A group's activity level varies, affecting both the developers' propensity to write to the mailing list and to contribute to source-code files. If our model approximates the reality better than other assumptions, it will perform well in predicting the links that arise in a network, compared to models that involve more parameters or are over-simplified. In what follows, we explain how we translate the idea of overlapping dynamic groups into a statistical model and combine it with the observed data.

In link prediction, the probability of a link being there or not is usually expressed with a *score*. We denote the score for $z_{ijkt} = 1$ by θ_{ijkt}. The score is monotonically positively related to $P(z_{ijkt} = 1)$, that is, if θ_{ijkt} is large, then also the probability of i and j to connect via channel k at time t is large: $P(z_{ijkt} = 1) \sim \theta_{ijkt}$, where $\sim$ denotes a monotonic positive relationship. For simplicity, we assume that the monotonic relationship is linear, that is, $P(z_{ijkt} = 1)$ is a linear transformation of θ_{ijkt}.[3] When the probability of an edge to exist is known, the edges are stochastically independent, and the probability of all events happening at the same time is just the product of the probabilities of the single events happening. Therefore, conditional on knowing all interaction probabilities, the probability of observing the given entire network reduces to the product of the probabilities of the single edges. This *conditional independence* is expressed in the following equation [FH15; Hof09]:

$$P(\underline{\mathbf{Z}}|\underline{\Theta}) = \prod_{i \neq j} P(z_{ijkt}|\theta_{ijkt}), \tag{4.1}$$

where $P(\underline{\mathbf{Z}})$ is the probability of observing a specific network and its tensor $\underline{\mathbf{Z}} \in \mathbb{B}^{N \times N \times 2 \times T}$. Then, $\underline{\Theta} \in \mathbb{R}^{N \times N \times 2 \times T}$ is the fourth-order tensor that contains the scores θ_{ijkt}. $\prod_{i \neq j}$ denotes the product of all probabilities of all the links z_{ijkt}, $i \neq j$. The score θ_{ijkt} contains all known and unknown information about developers i and j that influences their interaction in medium k at time t. Examples include whether developers i and j are currently collaborating on a task, whether they are interested in the same topics, and how much they currently contribute to the project. As we cannot observe $\underline{\Theta}$, we make assumptions about its nature and try to infer it from the observed network. In likelihood maximization, we choose $\underline{\Theta}$ such that it maximizes the probability of observing the given network $\underline{\mathbf{Z}}$. If we can choose between two different sets of values $\underline{\Theta}$, we choose the set that leads to the higher probability $P(\underline{\mathbf{Z}}|\underline{\Theta})$, that is, the set of θ-values that would have most likely resulted in the observed network. This way, we can compare different theories on the emergence of a network by comparing which theory was most likely to produce the observed outcome—and performed best in link prediction.

Every link has an unobserved probability of being existent, as no pair of developers is equal to another pair. Consequently, without further restricting assumptions, we would have to estimate just as many θ-values as there are different entries in $\underline{\mathbf{Z}}$, that is, $((N-1) \cdot N \cdot K \cdot T)/2$.[4] We need some restricting assumptions on the structure of the scores to reduce the number of parameters in the model. Moreover, restrictive assumptions improve the model's predictive power, as they reduce the influence of chance on the parameter estimation. The assumptions are equal to different models that describe developer behavior, and the one that best fits the observed data will be considered the most likely model in our analysis.

A relatively simple assumption to combine with our statistical rule is that the same developers that communicate or co-edit in time range t will communicate or co-edit in time range $t+1$ again. This assumption is naïve, as there is developer turnover in OSS projects, and developer participation depends also on other factors such as the release cycle of the project [FPB+15; KAD+15; LRS17]. Moreover, working on different features could also end up in changing interactions with different developers. Nevertheless, for simplicity, we will use this simple assumption on sustained developer interactions as a baseline model: If more complex models (which we will introduce later on) do not perform better than this most simple model, we will conclude that the more complex models do not add explanatory value. For baseline model **NAIVE**, assume that we have observed the network up to time range T, and want to look h time ranges into the future. Let $\hat{\theta}^{\text{NAIVE}}_{ijk,T+h|T}$ designate the estimated score for

3 An alternative would be to assume a logistic relationship. Such change would complicate the inference of model parameters but not change the model's basic intuition. See Hoff [Hof09] for an explanation in a matrix context.
4 Remember that $\underline{\mathbf{Z}}$ is symmetric, and z_{iikt} is not defined.

time range $T + h$ with information up to time range T. Then, the estimated score at time $T + h$ is equal to the observed value at time T:

$$\hat{\theta}^{\text{NAIVE}}_{ijk,T+h|T} = z_{ijkT} \tag{4.2}$$

For baseline model NAIVE, we do not assume stable long-term cohesion. Instead, we assume short-term alliances between developers that finish the tasks and discussions they have started. Consequently, the naïve model says that there is perfect stability of inter-personal links from time t to $t + h$, but no group behavior or long-term stability of interaction. We will compare this model's performance with the performance of a model that assumes that developers have a longer memory and that they are more likely to interact, the more often they have interacted before. To express this idea, we introduce a second baseline model, model **SUM**. Here, the score of two edges is equal to the sum of all edges:

$$\hat{\theta}^{\text{SUM}}_{ijk,T+h|T} = \sum_{\tau=1}^{T} z_{ijk\tau} \tag{4.3}$$

The scores in this model can be any integer from 1 to T if two developers have communicated or co-edited previously, else, it is 0. Two developers have a positive probability to co-edit if they have co-edited before, no matter *when* they co-edited.

As our baseline models NAIVE and SUM are rather simple, one could also come up with more sophisticated baseline models. For example, instead of simply summing up the previous edges, one could up-weight edges present in more recent time ranges or down-weight edges present only in older time ranges. However, such considerations would give rise to many additional questions (e.g., how many time ranges to consider and how to determine which weights would be appropriate), which would need to be explored and evaluated separately. The models we propose next are independent of such questions.

4.2.5 The Reduced-Rank Latent Factor Model

Neither of the two baseline models NAIVE and SUM pays particular attention to similar interests among developers, sub-groups, congruence of communication and co-editing, or time effects arising from long-term tasks and stable social relationships. Therefore, even if we assumed the conditional independence for these models, we would likely be wrong, as there are effects of reciprocity and other phenomena that disturb the assumption of independence in the model. To address RQ_1 (i.e., whether there are stable group structures in OSS projects), we introduce a model that translates the assumption of overlapping groups and a core-periphery structure into a statistical network model—the *reduced-rank latent factor model* (see Section 4.1.2 for the corresponding statistical background). We will evaluate this model later against the baseline models NAIVE and SUM by checking which one provides better predictions and fit.

Recall our assumption from Figure 4.1: We assume that developers work in overlapping temporal groups that are driven by tasks. These tasks affect the probability of edges among particular developers in a specific channel k at time t. We can restrict θ_{ijkt} such that it reflects the idea of overlapping dynamic sub-groups. One way to do so is to express θ_{ijkt} as a sum of *latent factors* that consist of three different elements:

(1) Assume that a project involves programming a feature. We call the creation of this feature "Task 1". Depending on her motivation and knowledge, some developer i may consider contributing to "Task 1". Denote her motivation/interest in "Task 1" by $u^k_{i1} \in \mathbb{R}$, where the superscript k means that we look at sub-groups only within communication (M) or co-editing (C) separately. If developer i is indifferent to working on "Task 1", then $u^k_{i1} = 0$. If she has an aversion against working on "Task 1", or if she is currently busy in a different task, $u^k_{i1} < 0$. If she is interested, $u^k_{i1} > 0$.
(2) The probability of developers i and j to interact is monotonically related to the product of u^k_{i1} and u^k_{j1}. If developer i is indifferent, $u^k_{i1} = 0$, then "Task 1" has no influence on her likelihood to co-edit or communicate with j on a task, as any number u^k_{j1} multiplied by $u^k_{i1} = 0$ is 0. If developer i is interested, and j has a strong aversion against task "Task 1", then they have u values with opposed signs. Then, $u^k_{i1} \cdot u^k_{j1} < 0$, and "Task 1" decreases their likelihood to interact.
(3) The task is worked on in time ranges 20 to 25. Let d^k_{t1} be the importance of "Task 1" at time t. Then, $d^k_{20,1}, \dots, d^k_{25,1}$, the *dynamic weight of "Task 1" from time range 20 to time range 25*, is relatively large.

Using these three insights, we assume that "Task 1" can be fully described by $\mathbf{u}^k_{r=1} \in \mathbb{R}^N$ and $\mathbf{d}^k_{r=1} \in \mathbb{R}^T$: These vectors contain the N developers' and T time ranges' values $u^k_{1,1}, \dots u^k_{N1}$ and $d^k_{1,1}, \dots, d^k_{T1}$ regarding "Task 1". As any multiple of Θ is equally valid as a score, $\mathbf{u}^k_{r=1}$ and $\mathbf{d}^k_{r=1}$ are not *identified*, that is, there are several equally valid solutions for the concrete values in the vectors. One way to fix the values is to restrict the vectors to length 1, $|\mathbf{u}^k_{r=1}|_2 = 1$ and $|\mathbf{d}^k_{r=1}|_2 = 1$, where $||_2$ is the Euclidean norm. "Task 1" is the first *latent factor*, that is, the first unobserved reason why developers i and j communicate and co-edit at a specific time. However, there may be more than one task. If there is a second task, we need to give a weight to both tasks to fix the importance of the tasks. Let the weights of "Task 1" and "Task 2" be λ^k_1 and λ^k_2. Then, the score is $\theta_{ijkt} = \lambda^k_1 \cdot u^k_{i1} \cdot u^k_{j1} \cdot d^k_{t1} + \lambda^k_2 \cdot u^k_{i2} \cdot u^k_{j2} \cdot d^k_{t2}$. Say that in total, there are $R \in \mathbb{N}$ latent factors or tasks. The score of two developers in a specific interaction channel at a specific time is determined by the sum of the R factors. This results in model **3D**, which uses a third-order tensor decomposition with rank reduction, looking at the information of a specific interaction channel:

$$\theta^{\text{3D}}_{ijkt} = \sum_{r=1}^{R} \lambda^k_r \cdot u^k_{ir} \cdot u^k_{jr} \cdot d^k_{tr} \tag{4.4}$$
$$\text{with} \quad \theta^{\text{3D}}_{ijkt}, \lambda^k_r, u^k_{ir}, u^k_{jr}, d^k_{tr} \in \mathbb{R}, \quad r \in \{1, \dots, R\}, \quad R \in \mathbb{N}.$$

Developer i can be described by R scores that describe how involved he or she is in task $r \in \{1, \dots, R\}$. These R values are encoded in a vector $\mathbf{u}^k_i \in \mathbb{R}^R$. The same is true for developer j. This common interest is multiplied with a time weight, which we call $\mathbf{d}^k_t \in \mathbb{R}^R$. Every factor r influences N developers and T time ranges. Let $\mathbf{u}^k_r \in \mathbb{R}^N$ be the vector of all developers regarding the r-th latent factor and $\mathbf{d}^k_r \in \mathbb{R}^T$ be the dynamic weights of the r-th latent factor within interaction channel k.

We can use this model for link prediction, by assuming that the future importance of tasks at time $T + h$ is likely to be equal to the importance of tasks in the current time range T. Should this model perform better than the baseline models, there is evidence for group behavior, but no evidence about the stability of the group importance over time yet. Note that more than one of the tasks can currently be important, so we have overlapping group behavior. The current level of activity of the overlapping groups describes the project at time T. Future activity levels are most likely to be equal to the current weights of the factors:

$$\hat{\theta}^{\textsc{3d}}_{ijk,T+h|T} = \sum_{r=1}^{R} \lambda_r^k \cdot u_{ir}^k \cdot u_{jr}^k \cdot \hat{d}^k_{T+h,r|T} \tag{4.5}$$

We can now make an assumption on the future importance of the r-th task. In model 3D, we assume that the importance of the component will stay the same as it is currently:

$$\hat{d}^k_{T+h,r|T} = d^k_{Tr} \tag{4.6}$$

The model has an additional advantage compared to NAIVE and SUM: It increases the weight of developers with many edges to other developers by concentrating on the most influential patterns in the data. When we infer only a few factors from the observed data (R is small), then the model is forced to focus on meaningful coalitions and groups. Concentrating on fewer groups reduces the risk of over-interpreting spurious patterns. This reduced risk, together with the relatively low number of parameters, improves interpretability and link prediction performance. The model concentrates on the more "important" developers, where "important" has a multi-layer interpretation: If developer i (a) interacts with a large number of other developers, (b) over several time ranges, she receives higher weights in the developer effects $\mathbf{u}_i^k$. Her scores will likely be higher—but the single score θ_{ijkt} also depends on $\mathbf{u}_j^k$, the scores of the potential partner. The rank reduction "considers" occasional interaction to be "noise" and reduces their weight in the prediction. This noise reduction is useful for prediction.

4.2.6 Congruence of Sub-Groups in Coordination and Programming Work

Previous research has shown that, in OSS projects, core developers often take over both the bulk of communication and programming work [CH08; JSW11; MFH02; TRC10]. This brings us to RQ_2, in which we ask whether communicative tasks and co-editing tasks result in the same group structures. If there is no distinction between both types of tasks, then the sub-groups should be similar in both interaction channels, since the "mirroring hypothesis" states that there shall emerge joint group structures due to a socio-technical congruence between communication and programming activities. Estimating only one group structure for both interaction channels may reduce noise and estimation uncertainty even more, and it helps to measure the congruence of group structures in both channels. Thus, we assume that the developer i's interest in a task is the same for communication and co-editing (u_{ir}^k simplifies to u_{ir} in both channels). Moreover, when estimating only a joint group structure for both communication and co-editing, we assume that a task is similarly important for both communication and co-editing at time t. So, we estimate only one weight for latent factor r at time t, instead of separate weights for communication and co-editing (d_{tr}^k simplifies to d_{tr}). Also the relative importance of the tasks is assumed to be the same (λ_r^k simplifies to λ_r). Instead, we assume that a task can require more or less communication and co-editing and add a c_{kr} parameter to the equation.

Let $c_{k1} \in \mathbb{R}$ be the channel effect of "Task 1". Assume that, to fulfill "Task 1", developers have to co-develop code, but no communication is necessary. Then, let $c_{M1} = 0$, and $c_{C1} > 0$.

Similar to the previous specifications, there is a c_{kr} for every task r, and $\mathbf{c}_{r=1} \in \mathbb{R}^2$ are the two interaction channels' strengths of relationship to the task $r = 1$. The length of the vector is, again, normalized: $|\mathbf{c}_{r=1}|_2 = 1$. Model **4D** extends model 3D by one more dimension:

$$\theta^{4D}_{ijkt} = \sum_{r=1}^{R} \lambda_r \cdot u_{ir} \cdot u_{jr} \cdot c_{kr} \cdot d_{tr} \tag{4.7}$$
$$\text{with} \quad \theta^{4D}_{ijkt}, \lambda_r, u_{ir}, u_{jr}, c_{kr}, d_{tr} \in \mathbb{R}, \quad R \in \mathbb{N}.$$

$$\hat{\theta}^{4D}_{ijk,T+h|T} = \sum_{r=1}^{R} \lambda_r \cdot u_{ir} \cdot u_{jr} \cdot c_{kr} \cdot \hat{d}^{4D}_{T+h,r|T}, \tag{4.8}$$
$$\hat{d}^{4D}_{T+h,r|T} = d_{T,r} \tag{4.9}$$

The parameter c_{kr} describes the effect of a task r on the interaction channel k, that is, whether the task requires a lot of cochange activity or mail activity. θ_{ijkt} can now also be higher when a task affects multiple (in our case, two) interaction channels.

The core assumption of the low-rank factor model is the separability of the three forms of dependency. *Inter-personal dependency* describes relationships and groups among developers, which are due to common interests, common tasks, or other unobserved developer characteristics, denoted by $\mathbf{u}_r$. *Temporal dependency* describes that developers are likely to work on specific topics at a specific time. The static aspect of "Task r" is described by user effects and channel effects. $\mathbf{d}_r$ denotes the current importance of "Task r". This separability assumption is less ambitious than the models above but not harmless: When the requirements of a task change over time, or when groups change only incrementally over time, the model is likely not to describe the real situation well. *Cross-channel dependency* describes the integration of co-editing and communication interaction channels. Independently of who contributes and independently of time, the task requires a certain amount of communication and co-editing. $\mathbf{c}_r$ denotes this effect of the task on the interaction channel. In other words, the model is based on fixed inter-personal relationships and cross-channel patterns, and allows only the current importance of these patterns to change. If one task is done, the respective factor's dynamic weights will revert to 0. Changing coalitions manifest only in changes in the weights of the groups. The group structures themselves are assumed to be constant in time. While this limits the applicability of the model for tracking changes in a group, it is useful for identifying groups and helps in link prediction due to the noise reduction effect.

4.2.7 Spectral Growth

d_{tr} in model 4D, as well as d^k_{tr} in model 3D, describe the current activity level of latent factor r at time t. So far, we assumed that in the future, this importance will be constant: $\hat{d}_{T+h,r} = d_{Tr}$ and $\hat{d}^k_{T+h,r} = d^k_{Tr}$. However, if factor r has been growing more important in the last few time ranges, it can be a reasonable assumption that the group's importance will further increase in the next time ranges. Therefore, the trend in $\mathbf{d}_r$ can be extrapolated to the future, to make a *dynamic forecast*. This can be used to address RQ_3, in which we ask whether considering past activity in co-editing or communication does improve the prediction of future activities.

The assumption of constant factorial patterns with trends in dynamic importance has been discussed by Kunegis et al. [KFB10] under the name of "spectral stability" (see Section 4.1.2). They proposed to infer $\mathbf{u}_1, \ldots, \mathbf{u}_R$ from two consecutive time slices of a network (via matrix decomposition), and to extrapolate the changes in the dynamic weights to predict future links. According to Kunegis et al. [KFB10], the more stable the inter-personal alliances, the better an extrapolation of $\mathbf{d}_r$ to the future performs for link prediction. Suppose we want to predict developer interaction h time ranges ahead (h stands for forecast *horizon*). For a window length of 3 months, $h = 1$ is equivalent to predicting interaction within the next 3 months. Consequently, $h = 5$ is equivalent to predicting interaction in the 3-months window that lies 1 year in the future. We can make a prediction at any time range t, but use only information from the past. Therefore, $d_{T+h,r}$ is the unobservable weight of component r at time $T + h$, whereas $\hat{d}_{T+h|T}$ denotes the predicted value of $d_{T+h,r}$ given all information at time T. Kunegis et al. [KFB10] calculate d_{tr} and $d_{t-1,r}$ by matrix decomposition and vector matching, and calculate the growth of a latent factor as $\Delta d_{tr} = d_{tr} - d_{t-1,r}$. Assuming that this is the current change in the importance of latent factor r, they assume that this change will repeat itself for each of the h steps we go into the future:

$$\hat{d}_{T+h,r|T} = d_{Tr} + h \cdot (d_{Tr} - d_{T-1,r}) = d_{Tr} + h \cdot \Delta d_{Tr} \tag{4.10}$$

As we base our methodology on tensor decomposition, we can profit from more information and adapt the extrapolation strategy in several regards. First, we want to make use of data from more than one time range ago, that is, we want to use the information in the vector $\mathbf{d}_r \in \mathbb{R}^T$. Second, we have enough information to infer a "current trend" and a "current level" that we can extrapolate to the future. To estimate d_{t+h} with all information up to time range t, that is, to calculate $\hat{d}_{t+h|t}$, we use a double exponential smoothing. This model bases on the state-space model for univariate time series proposed by Hyndman et al. [HKS+02] (see Section 4.1.2), which estimates that current information is more valuable than old information for the prediction of a time series. This estimation is in line with our assumptions on the collaboration of developers, as knowing about the current collaboration of developers might be more valuable for predicting future collaboration than knowing that certain developers had collaborated several years ago. This means that d_T receives a higher weight than d_1 when predicting d_{T+h}. To avoid overfitting and vulnerability to small T, we use an additive double exponential smoothing extrapolation (see Hyndman et al. [HKS+02]).

We treat $\mathbf{d}_r = d_{1r}, \ldots, d_{tr}, \ldots, d_{Tr}$ as a univariate time series. This time series can look like the green solid line in Figure 4.3, which indicates that the importance of component r (e.g., a certain task shared among the developers) increased until time range 15, and then declined. At time T, the time series has reached a kind of stable level, with a slightly decreasing trending behavior. We call the current level of the time series at time T l_T, and we call the slightly decreasing trend b_T. When predicting the future of this time series, we extrapolate the current behavior to the future and assume that the level and trend will continue. The predicted value of d_{T+h} is denoted by $\hat{d}_{T+h,r|T}$, the estimated value with all information available at time T. We call the corresponding models 3D-EXT and 4D-EXT for "extrapolated": The predicted (extrapolated) values can be multiplied with the values u^k_{ir} and u^k_{jr} in model 3D-EXT, and u_{ir}, u_{jr}, and c_{kr} in model 4D-EXT, respectively, to obtain an estimate for future scores of the

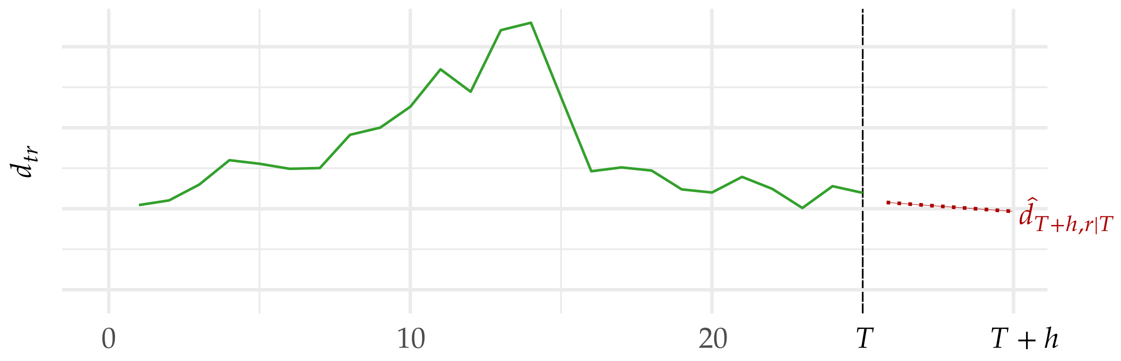


Figure 4.3: The current trend of the weights of the r-th factor (e.g., a certain task shared among the developers) is extrapolated to future time ranges for prediction.

developer pairs. Using 3D-EXT, we aim at answering RQ$_3$, whether past activity in co-editing or communication, respectively, improves the prediction of future co-editing or communication when compared to the simpler model 3D. Using 4D-EXT, we investigate whether the prediction on one communication channel can be improved by incorporating past activity on the other channel, as compared to the simpler models such as 4D.

The combination of $\mathbf{d}_r$ with a time-series smoothing method for extrapolation to the future reduces the vulnerability of the method to short time windows and provides additional stability. Combining the two methods—tensor decomposition and the state-space model—has been shown to be effective for link prediction [Sch22]. When we observe new information, that is, as soon as $d_{t+1,r}$ becomes observable, we can update our beliefs about the level and the trend of the time series $d_{t,r}$. Let ϵ_{t+1} describe the difference between our predicted value $\hat{d}_{t+1,r|t}$ and the observed $d_{t+1,r}$. We react to the mistake we made by updating $l_{t,r}$ to $l_{t+1,r}$ and by updating $b_{t,r}$ to $b_{t+1,r}$. Two parameters determine the strength of the adaptation of the estimate to new information, α_r and β_r. α_r describes how strong our adaptation of the level $l_{t,r}$ is, and β_r describes how strong our adaption of the trend $b_{t,r}$ is. Below, our updating equations and the prediction for a factor r's weight at time $t+h$ with information from time t are given:

$$l_{t,r} = l_{t-1,r} + b_{t-1,r} + \alpha_r \epsilon_{t,r}, \tag{4.11}$$
$$b_{t,r} = b_{t-1,r} + \alpha_r \beta_r \epsilon_{t,r}, \tag{4.12}$$
$$\hat{d}_{t+h,r|t} = l_{t,r} + h \cdot b_{t,r} \tag{4.13}$$

$\hat{d}_{t+h,r|t}$ is the predicted future weight of the r-th factor, assuming that the current trend $b_{t,r}$ and the current level $l_{t,r}$ describe the future dynamics of the r-th factor. This estimated $\hat{d}_{t+h,r|t}$ can then be put back into the prediction formulas for models 3D-EXT and 4D-EXT.

In model **3D-EXT**, this results in the following prediction equations for the prediction of the scores $\theta_{ijk,T+h}$ at the last observed time range T:

$$\hat{\theta}^{\text{3D-EXT},k}_{ijk,T+h|T} = \sum_{r=1}^{R} \lambda^k_r \cdot u^k_{ir} \cdot u^k_{jr} \cdot \hat{d}^{\text{3D-EXT},k}_{T+h,r|T}, \tag{4.14}$$
$$\hat{d}^{\text{3D-EXT},k}_{T+h,r|T} = l^{\text{3D-EXT},k}_{Tr} + h \cdot b^{\text{3D-EXT},k}_{Tr} \tag{4.15}$$

As the R groups are estimated separately for the *mail* and *cochange* channel, also the prediction for the future weights of the R groups are computed separately.

In model **4D-EXT**, there are R groups estimated for both channels. So, the dynamic of every group is the same in both channels:

$$\hat{\theta}^{\text{4D-EXT}}_{ijk,T+h|T} = \sum_{r=1}^{R} \lambda_r \cdot u_{ir} \cdot u_{jr} \cdot c_{kr} \cdot \hat{d}^{\text{4D-EXT}}_{T+h,r|T}, \tag{4.16}$$

$$\hat{d}^{\text{4D-EXT}}_{T+h,r|T} = l^{\text{4D-EXT}}_{Tr} + h \cdot b^{\text{4D-EXT}}_{Tr} \tag{4.17}$$

Model 3D-EXT assumes that the two modes of interaction have separate groups and group dynamics. Instead, model 4D-EXT assumes that both modes of interaction share a joint group structure. This assumption is based on the “mirroring hypothesis” [CB16], saying that there is a socio-technical congruence between organizational structures (e.g., communication) and structures arising from programming activities, which results in joint group structures.

The extrapolation of the current changes in the group weights brings additional parameters α_r and β_r that need to be estimated and can bring additional uncertainty to the estimations. This additional uncertainty can be harmful especially for small T, as then the parameters have to be estimated from a small sample size. However, it can also lead to improved predictions by better assessing recent group dynamics. It can predict growth where group weights are currently growing, and shrinkage where group weights are currently shrinking.

4.2.8 Summary of the Proposed Models

Every pair of developers has an individual probability of communicating or co-editing in a given time range. To estimate these probabilities, we need to restrict the number of parameters—especially if we want to predict future developer interactions. The insight that overlapping sub-groups in developer networks arise can be translated into a *dynamic low-rank factor model* [AGH+14a; Hof09] that bases on the assumption that co-editing and communication are driven by unobserved “tasks” or groups, which lead to developer clustering as well as cross-channel and dynamic dependency. Thus, we map interaction data to a higher-dimensional tensor structure and decompose it into time, person, and channel-related patterns via higher-order tensor decomposition [SCA+12]. The tensor structure retrieves group structures from communication and co-editing without aggregating both modes of interaction.

Deprived of the possibility to do classical statistical hypothesis testing, we base our validation strategy on related measures of predictive performance: We quantify temporal stability by measuring how much information on past collaboration events increases the performance of a prediction of future developer interaction. Furthermore, we investigate whether information on communication is useful for predicting co-editing and vice versa. Our method reduces the complexity level of group dynamics to two dimensions and allows us to visualize the dynamic group structures in simple plots.

We will compare our proposed models to verify the value of historical information and the group model for understanding the data structure. In Table 4.1, we summarize the six models and prediction strategies (including the two baseline models NAIVE and SUM). Models 3D, 4D, 3D-EXT, and 4D-EXT are based on the assumption of unobserved, independent latent factors that explain the correlation structure among the interaction events. These latent factors

Table 4.1: Overview of our models for performance comparison.

Model	Description	Equations
NAIVE	Developers who interact in channel k at time t are likely to interact at time $t+1$ in channel k again.	(4.2)
SUM	All information from time ranges $1, \dots, t$ at equal importance, developers who collaborated in three time ranges are three times as likely to collaborate again in the future compared to developers who collaborated in only one time range.	(4.3)
3D	Developer interaction is driven in both channels independently by dynamic, non-overlapping group behavior (also described above as "tasks"). The current activity level of a single group in one mode of interaction will be the same in the next time range.	(4.4), (4.5), (4.6)
4D	Developer interaction is driven in both channels simultaneously by dynamic, overlapping group behavior (also described above as "tasks"). The current activity level of a single group will be the same in the next time range.	(4.7), (4.8), (4.9)
3D-EXT	Based on 3D. However, activity levels of groups in co-editing *or* communication context have trends and levels, and a time-series model can extrapolate these trends to future time ranges.	(4.14), (4.15)
4D-EXT	Based on 4D. However, activity levels of groups have trends and levels, and a time-series model can extrapolate these trends to future time ranges.	(4.16), (4.17)

can be seen as "tasks", but are not limited to tasks: They also cover dynamic patterns of homophily and other reasons for inter-personal relationships. It does not matter whether it is a function that needs to be programmed, a discussion on a specific topic, or some other reason for co-editing or communicating. The latent factor is just some unobserved phenomenon that drives co-editing and communication activity of the developers, and that will manifest in the network topology and the tensor $\underline{\mathbf{Z}}$ [KFB10].

So, how are tensor decomposition and extrapolation linked to verifying stability in group structures among developers? We base our empirical validation strategy on four arguments:

(a) If the factorial structure adequately reflects real-world group structures, we should see major events like the drop-out of an important developer disturb the project in the decomposition. The decomposition attributes high weights to developers who are well integrated into the community structures of the project. Hence, the drop-out of a well-connected developer can—in a very active group—lead to the identification of a completely separated new group after the drop-out.

(b) If there are sub-groups in the network that describe the social network structure reasonably well (see RQ_1), then a latent factor model (3D, 4D, 3D-EXT, and 4D-EXT) should perform better in link prediction than NAIVE and SUM. The reason is that not very well integrated developers are not identified to be part of the important sub-groups and are attributed a minor role in the prediction. Should the contribution behavior of these not very well integrated developers be just as stable and continuous as the identified group structures, we will not see a benefit for the models 3D, 4D, 3D-EXT, and 4D-EXT: These models attribute a weight to developers according to their position in the network—developers who are integrated into important sub-groups receive a higher weight than peripheral developers, who contribute only temporarily and add little contribution to the project. The two baseline models, instead, do not take a developer's position in the network into account and only count the absence or presence of an interaction with another developer.

(c) If the same sub-groups that exchange e-mails do also collaborate in programming (see RQ_2), then 4D-EXT should perform better than 3D-EXT, and 4D should perform better than 3D. When the group structures are congruent in both interaction modes, the estimation will be more reliable in the models 4D-EXT and 4D than in their three-dimensional counterparts as more evidence on the groups is provided.

(d) If the alliances are of changing importance (see RQ_3), then the weight extrapolation (3D-EXT and 4D-EXT) should perform better than a non-dynamic prediction method (3D and 4D). The state-space time-series prediction method allows the groups to continue growing or shrinking over time, whereas the models without a time-series component are more vulnerable to temporarily different weights in the last time range and assume constant importance of tasks or groups.

The predictive performance of the models will help us to judge the stability of sub-group importance and collaboration dynamics. The insights of the decomposition reveal the social integration of developers and the changes in social relationships, as opposed to vertex centrality measures or simply event counts. By validating the fit of an assumption with the predictive accuracy of the corresponding model, we proceed like Dong et al. [DTW+12]: They use the assumption that people engage similarly in multiple social contexts as an argument to estimate similar interaction structures for link prediction in these different contexts. Then, they interpret the improvement of the link prediction as evidence for the correctness of the assumption.

Concerning argument (a), we will look for significant events that involved a drop-out or change of core developers and see whether these events have a corresponding manifestation in the factor structure. Concerning argument (b), we will use the rank reduction for link prediction and verify whether the reduced-rank prediction performs better than the non-reduced models. Concerning argument (c), we will compare the predictive performance of the respective models on third-order tensors $\underline{\mathbf{Z}}_M$ and $\underline{\mathbf{Z}}_C$ against the predictive performance of the respective models on the fourth-order tensor $\underline{\mathbf{Z}}$. Concerning argument (d), we will compare the predictive performance of the rank-reduced model with extrapolation of group dynamics with simpler models that do not use the time-series state-space model.

4.3 Implementation and Study Design

Having laid out our models and our validation strategy, we will now proceed to its application to a series of subject projects. The main purpose is to demonstrate the practicability of our measurement and analysis method. We start with a detailed overview of how and from which sources we retrieve and operationalize data on communication and co-editing (Section 4.3.1). Then, we introduce some descriptive measures that help us in getting an overview of the interaction dynamics and the congruence of communication and co-editing (Section 4.3.2). Next, we explain how we retrieve the parameter estimates for the models 3D, 4D, 3D-EXT, and 4D-EXT with a canonical tensor decomposition (Section 4.3.3) and how we apply time-series extrapolation for the models 3D-EXT and 4D-EXT (Section 4.3.4). Finally, we discuss how to measure the performance of the models in time, and we verify the robustness of our models regarding discretionary decisions regarding the implementation (Section 4.3.5).

Table 4.2: Numbers of developers (N) in our 10 subject projects, with start of the first and end of the last 3-months time range, number of analyzed time ranges (T), and project domain.

Project	N	Start	End	T	Project Domain
Jailhouse	17	2013-11-20	2016-08-24	11	Hypervisor abstracting from hardw. & operating syst.
OpenSSL	153	2002-04-21	2016-02-19	55	Encryption library to secure Internet connections
BusyBox	217	2003-01-14	2016-02-16	52	UNIX command-line tool suite
ownCloud	471	2010-03-24	2018-05-20	32	File hosting service
QEMU	919	2003-04-29	2016-07-27	52	Virtual machine emulator
Git	943	2005-04-13	2017-03-12	47	Version-control system
Wine	1 092	2002-04-06	2017-11-16	62	Runtime env. for Windows programs on UNIX systems
Django	1 131	2005-08-01	2017-12-04	49	Web framework
FFmpeg	1 256	2003-01-06	2017-12-12	59	Audio and video encoding tool suite
U-Boot	1 356	2000-01-01	2017-12-18	71	Boot loader mostly used in embedded systems

4.3.1 Study Design and Implementation Overview

To evaluate our method on differently shaped projects and to demonstrate that and to which extent it is applicable, we apply our method to 10 well-established OSS projects from different project domains (see Table 4.2). Our subject projects cover a broad range of project sizes, ranging from a total of 17 individual developers for Jailhouse to 1 356 individual developers for U-Boot. All our subject projects used mailing lists as the main channel for coordination in the investigated time period, and most of them have strict regulations to submit patches to the mailing list for discussion before integrating them into the repository.[5] Hence, in all the projects, mailing lists are a historically rich and well-established communication channel.[6]

For this study, we extract commit data from Git repositories and e-mail data from the mailing-list archive Gmane, using the tools Codeface and nntp2mbox, which we have introduced in Section 2.4. We use only mailing lists that involve primarily developers, and not lists that are addressing users. Additional information regarding the downloaded mailing lists and the analyzed Git repositories are available on our supplementary website.[7] For the construction of the developer networks (i.e, cochange networks and mail networks), we use the library coronet, as explained in Section 2.4. Noteworthy, in this study, we consider e-mail communication only among developers that have also contributed to the source code. That is, we do not consider communication among users or between users and developers. Consequently, there may be developers in our networks who have edited source code but not communicated, but not vice versa. We use time windows of 3 months, indicating in a binary decision whether there has been *mail* interaction or not between vertices i and j, and whether there has been *cochange* interaction between vertices i and j. Meneely and Williams [MW11]

5 The only exception here is project Django, which uses pull requests, which are not discussed on the mailing list, instead of patch submissions to the mailing list. We include this project as it is a highly active project and we are interested whether the group structures arising from the discussions on the mailing list are, even though not containing patch discussions, similar to the group structures arising from co-editing activities.

6 This holds for all the analyzed time ranges of all our subject projects except for project ownCloud, which has abandoned the use of its mailing list in 2016. We discuss this issue for project ownCloud in Section 4.4.1.

7 https://se-sic.github.io/paper-groupdynamics-oss-tensordecomposition/

as well as Joblin et al. [JAH+17] have shown that 3-months time windows are a good aggregation level to describe topological features of socio-technical developer networks. We further count repeated *cochange* or *mail* contact within a 3-months time window as a single interaction. The developers (i.e., vertices) are tied to one another if they have edited the same file or responded to the same mailing-list thread within one time range. As described in Section 4.2.3, this results in a binary $N \times N \times 2 \times T$-dimensional array of two modes of interaction between N developers; $\underline{\mathbf{Z}} \in \mathbb{B}^{N \times N \times 2 \times T}$. As the collaborative editing of source code does not imply a natural direction for relationships, we use only undirected, unweighted, simplified networks. All preprocessing and analysis scripts are available on our supplementary website.[7]

4.3.2 Exploratory Congruence Analysis

To explore the data, we compare the activity level as the number of interaction events for both channels. Let N_t be the number of developers that have either communicated or co-edited at time t. Furthermore, let $n_{M,t}$ be the number of *mail* events at time t, and $n_{C,t}$ be the number of *cochange* events at time t:

$$n_{M,t} = \frac{1}{2} \sum_{ij} z_{ijMt} \tag{4.18}$$

$$n_{C,t} = \frac{1}{2} \sum_{ij} z_{ijCt} \tag{4.19}$$

The average numbers of *mail* and *cochange* events over all time ranges are defined as:

$$\bar{n}_M = \frac{1}{T} \sum_t n_{M,t} \tag{4.20}$$

$$\bar{n}_C = \frac{1}{T} \sum_t n_{C,t} \tag{4.21}$$

To make the measure comparable across projects, we normalize $\bar{n}_M$ and $\bar{n}_C$ with the number of potential edges in the network. The number of possible edges within a single channel of interaction is $\frac{1}{2}N(N-1)$, as the edges are undirected. We define the measures $\bar{n}_C^{\%}$ and $\bar{n}_M^{\%}$ and call them *average density* of the *mail* and *cochange* network, respectively:

$$\bar{n}_M^{\%} = \frac{\bar{n}_M}{\frac{1}{2}N(N-1)} \cdot 100 \tag{4.22}$$

$$\bar{n}_C^{\%} = \frac{\bar{n}_C}{\frac{1}{2}N(N-1)} \cdot 100 \tag{4.23}$$

To provide some first impressions about the *relationship* between *mail* and *cochange* events, we compute the number of edges that are present in both interaction channels at time t, and call this measure $n_{11,t}$, where $n_{11,t}$ is the number of edges that are present both in the *mail* network and in the *cochange* network at time t:

$$n_{11,t} = \frac{1}{2} \sum_{ij} (z_{ijMt} \cdot z_{ijCt}). \tag{4.24}$$

$n_{11,t}$ is bounded: Its minimum is 0 and its maximum is equal to the minimum of the two values $n_{M,t}$ and $n_{C,t}$. The maximum of $n_{M,t}$ and $n_{C,t}$ is the number of edges that can be present within a time range t and within a single channel: $N(N-1)/2$. If $n_{11,t}$ is close to its maximum, there is a high temporal congruence of interaction across channels. However, the measure depends on the number of vertices and interaction events.

To provide a more comparable measure for the congruence of *mail* and *cochange* events, we use the "phi"-coefficient ϕ [RN88] as a measure for the correlation of the binary events in the *mail* and *cochange* networks. As we measure it for every time range, we index it with t. To compute the measure, let $n_{00,t}$ be the number of events that are neither present in one nor the other channel. Similarly, $n_{01,t}$ is the number of edges that are only present in the *cochange* network, and $n_{10,t}$ is the number of events that is only present in the *mail* network.

$$n_{00,t} = \frac{1}{2} \sum_{ij} (1 - z_{ijMt}) \cdot (1 - z_{ijCt}) \tag{4.25}$$

$$n_{01,t} = \frac{1}{2} \sum_{ij} (1 - z_{ijMt}) \cdot z_{ijCt} \tag{4.26}$$

$$n_{10,t} = \frac{1}{2} \sum_{ij} z_{ijMt} \cdot (1 - z_{ijCt}) \tag{4.27}$$

$N(N-1)/2 - n_{M,t}$ is the number of "absent" *mail* events and $N(N-1)/2 - n_{C,t}$ is the number of "absent" *cochange* events at time t. Then, ϕ_t is the product of the "congruent" number of events minus the product of the "different" events, scaled by the square root of the number of all four groups of events:

$$\phi_t = \frac{n_{11,t} \cdot n_{00,t} - n_{01,t} \cdot n_{10,t}}{\sqrt{n_{M,t} \cdot n_{C,t} \cdot (\frac{1}{2}N(N-1) - n_{M,t}) \cdot (\frac{1}{2}N(N-1) - n_{C,t})}} \tag{4.28}$$

ϕ_t is equal to 1 if all the edges in *cochange* are present also in *mail* and vice versa (100% overlap), and -1 if none are present in both channels (0% overlap). It is equal to 0 if there is a random overlap between both groups, the measure can therefore be interpreted like a correlation (ϕ is equal to the Pearson correlation when the variables of interest are binary). The four measures of interest, $n_{M,t}$, $n_{C,t}$, $n_{11,t}$ and ϕ_t are visualized for every project, as for example in Figure 4.5. Only some of the plots (the best suited for explaining and interpreting our findings) are included in this chapter, the others can be found on our supplementary website. Our findings take all the plots and values into consideration, though. Additionally, we provide and discuss the time averages of every measure, $\bar{n}_M = \sum_t n_{M,t}/T$, $\bar{n}_C = \sum_t n_{C,t}/T$, $\bar{n}_{11} = \sum_t n_{11,t}/T$, and $\bar{\phi} = \sum_t \phi_t/T$ for every subject project.

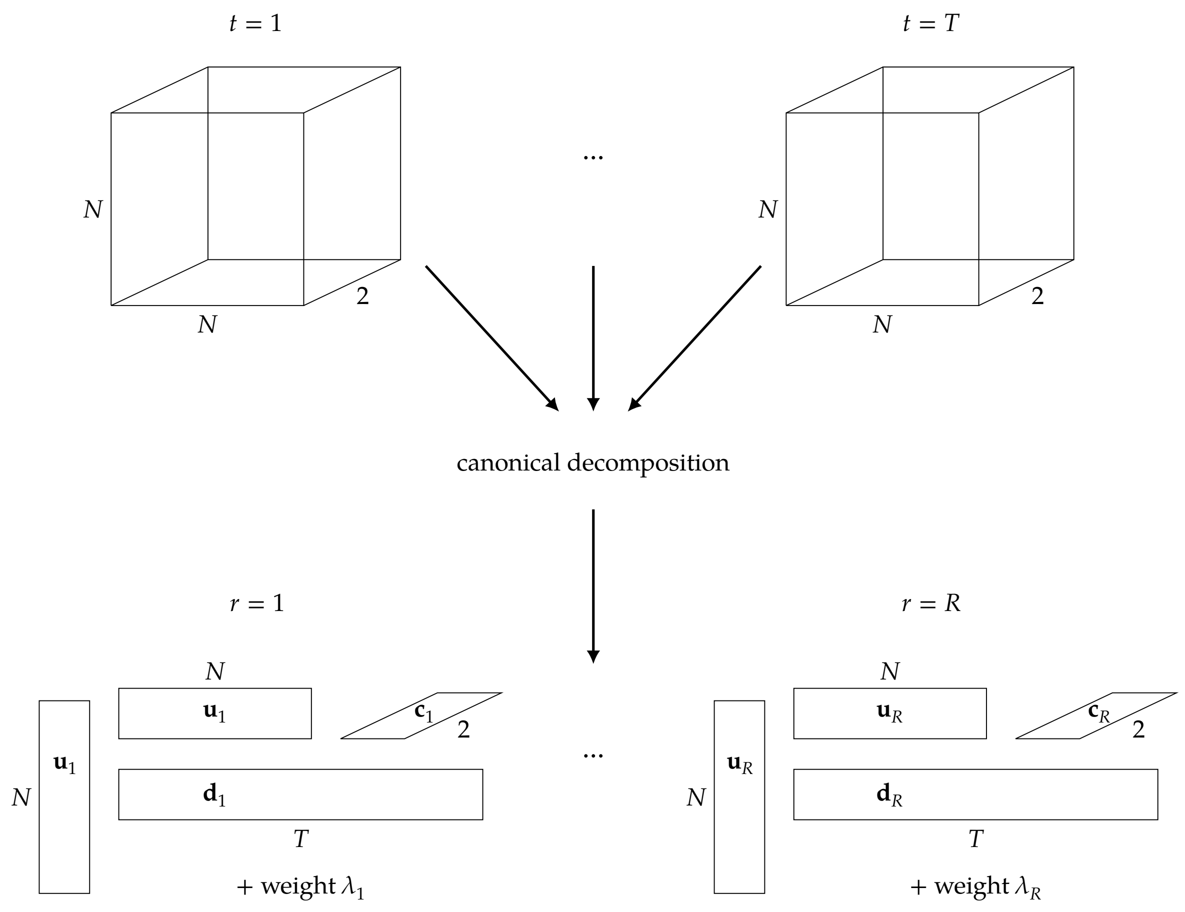


Figure 4.4: The canonical decomposition of $\underline{\mathbf{Z}}$: The four-dimensional data structure is reduced to R rank-one tensors, which are composed of the vectors $\mathbf{u}_r$, $\mathbf{c}_r$, and $\mathbf{d}_r$ of length N, 2, and T, respectively.

4.3.3 Canonical Tensor Decomposition

The multiplicative latent factor model translates the assumption of overlapping cross-channel dynamic groups (see Figure 4.1) into a statistical model. None of the parameters can be observed (which is why the factors are *latent*), but we can estimate them from the data with a tensor decomposition: $\mathbf{u}_r$, $\mathbf{c}_r$, and $\mathbf{d}_r$ can be inferred from the observed network $\underline{\mathbf{Z}}$: When the data are organized as a tensor $\underline{\mathbf{Z}}$, we can use a canonical decomposition to infer the individual latent factors from the observed interaction events [AGH+14b], as illustrated in Figure 4.4. Via canonical tensor decomposition, $\underline{\mathbf{Z}}$ is decomposed into R four-dimensional rank-one tensors, each consisting of the vectors $\mathbf{u}_r$, $\mathbf{c}_r$, and $\mathbf{d}_r$, and a scalar weight λ_r, where $r \in 1, \ldots, R$. In our implementation, we use the `cp()` function from the `rTensor` package [LBW15] for the canonical tensor decomposition.

4.3.4 Time-Series Extrapolation

After applying the canonical decomposition, we proceed to the extrapolation of trends in $\mathbf{d}_r$. For this purpose, we use function `holt()` from package `forecast` [HAB+18]. The function optimizes the parameters α and β autonomously.[8] As we assume that single latent factors describe individual groups or "tasks", treating each series of weights separately makes more sense than multivariate techniques. We do the estimation separately for the R latent factors. The exponential smoothing method is commonly used as a data-driven prediction method for univariate time series [HKS+02]. It is optimized to adapt to slowly-varying trends and changes in the level. Note that exponential smoothing can be inadequate when there are only few time ranges (small T), as it requires a certain burn-in period to perform well. If T is small, simpler models or a pre-specified α and β should be used. In our cases, the time series are reasonably long, and, therefore, we let the algorithm optimize the parameters by itself.

4.3.5 Cross-Validation

To validate changes in stability of the network's social structure, the predictive performance needs to be measured over time. We use a cross-validation technique inspired by time-series analysis and the *area under curve* (*AUC*) performance measure to make changes in the performance visible. AUC is the evaluation metric of choice when the analyzed link data are sparse.[9] Sparse means that there are less existing edges between vertices in a network than non-existing edges between vertices (e.g., when there are more zero elements in the corresponding adjacency matrix than non-zero elements). As there are only few edges in a sparse network, the prediction of the non-existence of edges between vertices does not really matter for the predictive performance, as we are interested into predicting edges, of which only few exist. AUC is indifferent to class imbalances of existing and non-existing edges in a network.

For validating predictive performance, we start the computation of predictions and performance measures with half of all available time ranges, that is, with a tensor of order $N \times N \times 2 \times \lceil \frac{1}{2}T \rceil$. Then, we extrapolate the time weights that result from this decomposition to $\lceil \frac{1}{2}T \rceil + h$, and measure predictive performance. We then extend the tensor by one time range. That is, at the second iteration, the tensor is of order $N \times N \times 2 \times (\lceil \frac{1}{2}T \rceil + 1)$. This cross-validation strategy adequately honors the dynamic structure of the data. While updating the tensor decomposition becomes advisable and even mandatory for larger tensors than the ones that we are dealing with [VVL17], our network sizes allow us to generate a new decomposition for every cross-validation iteration.[10]

Next, we choose a performance measure. The *receiver operating characteristic* (*ROC*) curve balances specificity (penalizes *false positives*) and sensitivity (penalizes "missed" positives, that is, *false negatives*) of a prediction method. The ROC curve depends on the *true-positive rate* (*TPR*) and the *false-positive rate* (*FPR*). This is advantageous in the case of sparse networks, where *true negatives* are of limited importance [ME11]. The ROC curve has been criticized for ignoring

8 The optimization uses the Akaike Criterion, for details see Hyndman et al. [HAB+18].

9 An explanation of why this is the case and of the intuition behind the AUC measure has been provided by Chen et al. [CHY+18].

10 We ignore z_{iikt}, that is, the main diagonal of the tensor slices $\mathbf{Z}_{..kt}$, as self-links are not defined. Nevertheless, the values θ_{iikt} for these links are positive, as u_{ir}^2 simply expresses the importance of developer i for component r.

the predicted probability values and the goodness-of-fit of the model [e.g., LJR08], and some researchers recommend the average precision rather than ROC and AUC [YLC15]. Still, ROC and AUC are the most widely accepted metric [CHY+18]. A single-number summary of the shape of the ROC curve is the *area under curve* (*AUC*). The expected AUC for random guessing is 0.5, and the lower and upper limit of the AUC are 0 and 1, respectively [Zhu04]. We compute an AUC measure for our models, separately either for the prediction of *mail* or for the prediction of *cochange* events. The performance of the models indicated by the AUC provides evidence on the beneficial effect of simultaneous consideration of co-editing and communication, of rank reduction, and of the extrapolation of trends in group weights. For the computation of the performance measures, we use the `roc()` function from the `pROC` package [RTH+11].

To aggregate the performance curves to a single number per project and prediction method, we average the performance measures and directly compare the methods. As we start the cross-validation at $\lceil \frac{1}{2}T \rceil$, and calculate it up to $T-h$,[11] the aggregated performance measure per subject project and method is:

$$\overline{\mathrm{AUC}} = \frac{1}{(\lfloor \frac{1}{2}T \rfloor - h)} \sum_{t=\lceil \frac{1}{2}T \rceil}^{T-h} \mathrm{AUC}_t \qquad (4.29)$$

We distinguish different prediction horizons for seeing at what forecast horizon h the rank reduction leads to the highest benefit compared to the baseline models. We expect the general performance to decrease with a growing forecast horizon h. However, we expect rank reduction to be beneficial for prediction as it decreases the noise in the data, and, therefore, we expect that the advantage of the rank reduction will show more in a 12-months-ahead than in a 3-months-ahead prediction.

We need to validate not only the performance in time, but also the performance for different values of R. As we lack a ground truth on the number of groups in the data, we need to assume that when the "right" groups have been identified, the predictive performance of the model is high. The risk of finding spurious patterns increases with R, and with it, the risk to find different interpretations for repeated estimation. Depending on whether the interpretation of group structures is very important, there are three possible approaches: (a) Checking the interpretation for repeated estimation with different seeds, (b) analyzing the variance in the time components, and (c) checking the predictive performance of the choice of R.

(a) If the interpretation of the patterns changes with the random seed, this is a sign for a too large rank R that leads to spurious patterns.
(b) The canonical decomposition does not treat the time dimension differently than the other dimensions. This means that in the "eyes" of the algorithm, t is as "close" to $t+1$ as to $t+5$. As we expect group constellations not to change too abruptly every time range, we are looking for an R value that produces relatively smooth time patterns in $\mathbf{d}_r$.
(c) To avoid overfitting looking for the perfect R, we check the robustness of the predictive performance of the choice of R by cross-validating R in the range between 1 and 20. We visualize the outcomes and discuss their implications for the project's social structure.

All three robustness checks are important to avoid interpreting spurious correlations. A small R can underfit, a large R can overfit the data. This has implications for the different use

11 Remember that, to measure the predictive performance for forecast horizon h, $\theta_{ijk,T+h|T}$, we can only use information up to h time ranges before the final time range.

Table 4.3: Characterization of *mail* and *cochange* activity in our 10 subject projects. N is the total number of developers involved in each project. n_{Mt}^{max} is the maximum number of *mail* edges per time range (Equation 4.18), $\bar{n}_M$ is the average of the number of *mail* edges over time (Equation 4.20), and $\bar{n}_M^{\%}$ is the average density of the *mail* network (Equation 4.22). The definitions are equivalent for n_{Ct}^{max}, $\bar{n}_C$, and $\bar{n}_C^{\%}$. $\bar{n}_{11}$ is the average number of intersecting edges (Equation 4.24), and $\bar{\phi}$ is the average ϕ_t coefficient (Equation 4.28).

	N	n_{Mt}^{max}	$\bar{n}_M$	$\bar{n}_M^{\%}$	n_{Ct}^{max}	$\bar{n}_C$	$\bar{n}_C^{\%}$	$\bar{n}_{11}$	$\bar{\phi}$
Jailhouse	17	27	11.8	8.69	27	9.8	7.22	4.6	0.39
OpenSSL	153	159	20.3	0.17	737	94.5	0.81	6.8	0.15
BusyBox	217	150	62.6	0.27	300	110.7	0.47	16.2	0.19
ownCloud	471	111	29.2	0.03	1 964	929.3	0.84	12.8	0.08
QEMU	919	1 651	723.3	0.17	9 888	2 586.3	0.61	368.0	0.20
Git	943	1 892	750.0	0.17	3 855	2 270.1	0.51	230.8	0.18
Wine	1 092	912	446.5	0.07	5 567	3 671.1	0.62	218.3	0.18
Django	1 131	266	131.7	0.02	9 370	1 991.5	0.31	48.4	0.18
FFmpeg	1 256	1 595	569.5	0.07	8 572	3 888.6	0.49	279.4	0.21
U-Boot	1 356	1 139	455.4	0.05	3 643	1 197.3	0.13	163.4	0.18

cases of the analysis: A large R can detect more granular changes in group constellations, thus, a large R can help to understand known problems in the past. However, a small R can be better for predicting future interaction, as it is less prone to over-interpret changes from the past.

4.4 Results

For the purpose of presenting our results, we mainly focus on a single case study:[12] BusyBox. This project was, based on chance, the first we investigated. We chose it for its relatively small size and number of developers. With this case study, we investigate the exploratory value of our method. We will then answer our research questions by comparing the models' predictive performance and generalize our findings over 10 subject projects. We start with an exploratory perspective on the data in Section 4.4.1. Then, we proceed with the description of the canonical decomposition in Section 4.4.2. The ensuing question is how much our results depend on the choice of R. Thus, we continue with the results of the cross-validation of R in Section 4.4.3. Then, we discuss the models' performance for a fixed R in Section 4.4.4. Finally, we summarize our results and answer our research questions in Section 4.4.5.

4.4.1 Descriptive Insights

To provide a first impression of the congruence of *mail* and *cochange* activity, we discuss the descriptive statistics of the activities in our subject projects that we show in Table 4.3.

Relationship Between the Number of Developers and Interaction Frequency First, the *mail* and *cochange* activity are not—as one might expect due to the quadratic relationship

12 Beside BusyBox, we also present a single figure on our subject project ownCloud to discuss exceptional cases. For visualizations of the results of our remaining subject projects, we refer to our supplementary website.

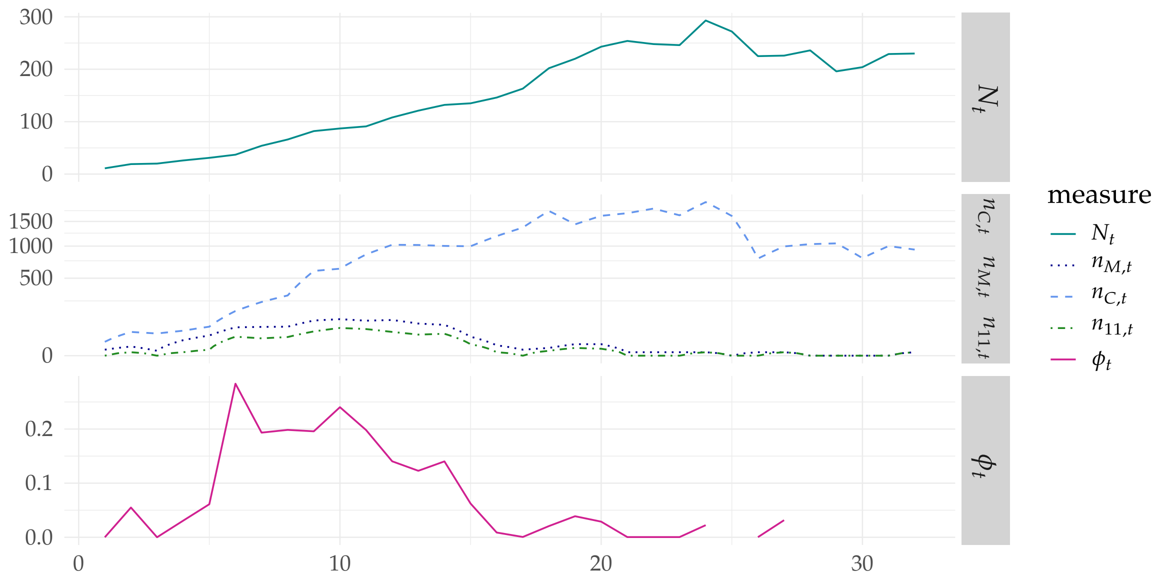


Figure 4.5: Current number of *mail* ($n_{M,t}$) and *cochange* ($n_{C,t}$) edges, correlation between *mail* and *cochange* ϕ_t, and number of active developers (N_t) for ownCloud over time. As the number of edges grows quadratically with N_t, we use a square root transformation for the y axis in the second panel. From time range 14 on, the mailing list seems to be abandoned. Accordingly, ϕ_t and $n_{M,t}$ as well as $n_{11,t}$ go down. When there is no mailing-list activity, ϕ_t cannot be computed.

between possible edges and vertices—monotonically related to the number of developers in a project. ownCloud is twice as large as BusyBox in terms of number of developers, and has fewer maximum and average *mail* edges, but roughly eight times as many *cochange* interactions, on average. There is very little *mail* activity in ownCloud and Django, with an average density of only 0.03% and 0.02%, respectively. Yet, Django has more than twice as many developers. With 0.84% average density in *cochange*, ownCloud seems to have a vivid co-editing culture, where many developers edit the same code files. ownCloud has little communication among the developers via the mailing list, though. It has a lower n_{Mt}^{max}, yet more than six times as many *cochange* edges n_{Ct}^{max} as BusyBox. Another project with very little density in the *mail* network is Django, which is the third largest project in our study, having 1 131 developers in total. For this project, also the density in the *cochange* network is lower than average. Another large project where both *mail* density and *cochange* density are low is U-Boot, with a $\bar{n}_M^{\%}$ of 0.05 and $\bar{n}_C^{\%}$ of 0.13. In general across all projects, there is approximately 30% to 150% more interaction per developer in the *cochange* network than in the *mail* network.

Relationship Between *Mail* and *Cochange* Modes ($\bar{\phi}$) The correlation between *mail* and *cochange* interaction is surprisingly stable across the projects, with values between 0.15 and 0.21. There is consistent evidence that communication activity on the mailing list is positively related to co-editing. Two exceptions apply: Jailhouse with its 17 active developers has a strong positive correlation (0.39) and ownCloud has a weak positive correlation (0.08). Together with the very low *mail* activity in ownCloud, this suggests that the mailing list is not strongly related to development activities here. In Figure 4.5, we can see that there is no

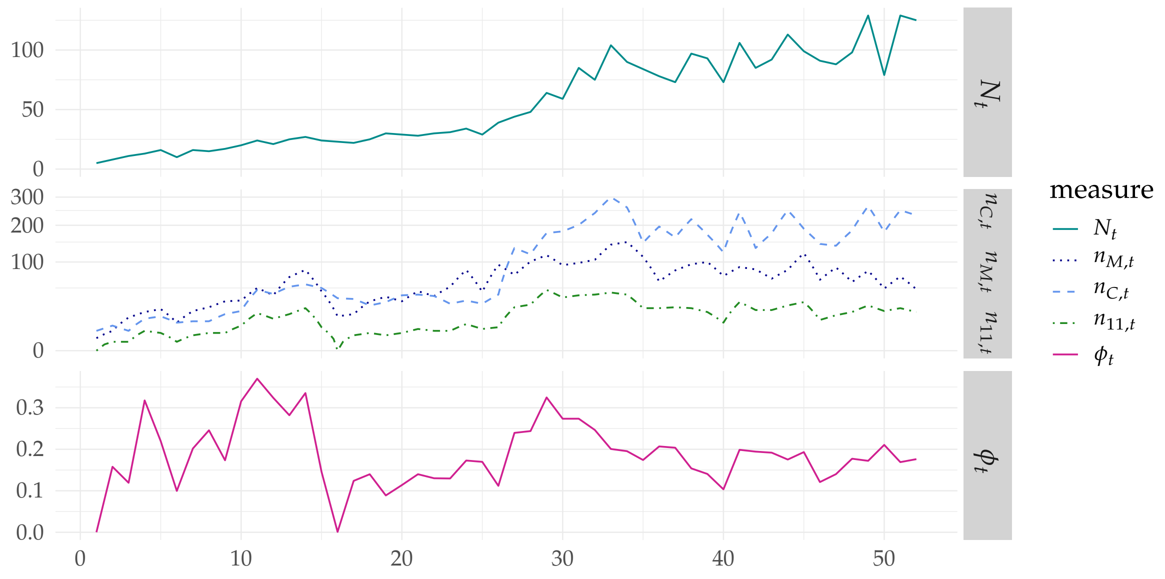


Figure 4.6: Current number of *mail* ($n_{M,t}$) and *cochange* ($n_{C,t}$) edges, correlation between *mail* and *cochange* ϕ_t, and number of active developers (N_t) for BusyBox over time. As the number of edges grows quadratically with N_t, we use a square root transformation for the y axis in the second panel. About one fourth of all *mail* edges is usually present in both channels. There are no common edges in $t = 16$ despite normal activity in both interaction modes.

ongoing activity on the mailing list from time range 14 on.[13] During the active usage of the mailing list (time ranges 6 to 14), ϕ_t was in the expected range. For Jailhouse, the smallest of our subject projects in terms of number of developers, the correlation is very strong. For Django, both the density of the *mail* network and the density of the *cochange* network are rather low. Nevertheless, the correlation is within the typical range of the subject projects.

Relationship Between Descriptive Measures and Project Phase In Figure 4.6, we show the evolution of the descriptive measures for BusyBox over time. In the early phases of the project (up until time range 30), the number of interaction events per developer grows quadratically with the number of active developers. After time range 30, the number of edges in the network and the number of simultaneous edges in both channels stagnate, despite continued growth in the number of developers. At the same time, ϕ_t is constant between 0.1 and 0.2. A maximum in the correlation of 0.31 at time range 29 happens in a phase of growing *mail* activity. N_t shows that BusyBox experiences strong growth from the 25th time range on. At the same time, $n_{M,t}$ and $n_{C,t}$ start growing, where *cochange* activity grows faster than *mail* activity. An interesting particularity of BusyBox can be seen in $n_{11,t}$ and, as a direct consequence, in ϕ_t: With the sharp drop of *mail* activity at time ranges 14–16, $n_{11,t}$ drops to 0: In time range 16, not a single edge is present in both channels—a sign for a major distortion around that time. Shortly before this drop, there has been a peak in *mail* activity. The distortion

13 This goes well with decisions of the ownCloud community to no longer use the mailing list: `https://central.owncloud.org/t/replace-mailing-lists/875/` (accessed at 2020-09-26). We, though, have decided to investigate this project as it has a comparably high amount of commit activity but a comparably low amount of e-mail activity (even before abandoning the mailing list). Therefore, we explore how our analysis method performs with respect to such anomalies in the data.

is also visible in ϕ_t: There is a moderate correlation between *mail* and *cochange* events, except for $t = 16$, where ϕ_t drops to 0. ϕ_t is pretty volatile for early phases of the project, but stabilizes to a value of between 0.10 and 0.20 in later time ranges. The phase of stability coincides with a rise in the number of active developers from time range 33 on.

General Observations Across All Projects When evaluating *all* projects in a similar way, we see that some projects show a strong relation between the general number of co-editing and communication events, whereas in others, both modes of interaction seem to be decoupled.

In the following list, we describe the findings for the projects in which *mail* and *cochange* activity seem to be relatively independent, and the *mail* activity does not follow general growth trends:

- For FFmpeg, the mailing-list traffic drops around time range 35. Only the number of *cochange* events grows as expected with the number of developers. This may indicate that only a constant subset of developers communicate via the existing mailing list after this change in trend behavior at time range 35. ϕ_t then stabilizes at a value of approximately 0.15, after a plateau of 0.25 before.
- In Wine, the number of edges in the *mail* networks slowly declines over time.
- For Django, the usage of the mailing list is constant along the whole time line. It is completely independent of the strong growth in the number of developers and *cochange* events. Accordingly, the more developers join, the smaller ϕ_t becomes. The *mail* network has very little in common with the *cochange* network from there on.
- For project U-Boot, the pattern is similar as for BusyBox and FFmpeg: The number of *cochange* events closely quadratically follows the number of developers, whereas the number of *mail* events stagnates.

In other projects, we observe that both means of interaction (*mail* and *cochange*) are closely related to each other:

- For Jailhouse, the mailing-list activity closely follows general growth trends. The correlation between both channels of interaction is strong. However, the maximum ϕ_t of 0.8 occurs at a time when there are only 4 developers in the project—of the 9 *cochange* events at that time, 8 (and only these 8) edges are also present in the *mail* network. The number of actively co-editing developers never exceeds $N_t = 13$.
- In OpenSSL, *mail* and *cochange* activity follow equally the general trend in the number of developers—the number of *cochange* edges follows more directly while the number of *mail* edges leaks a bit behind. Due to the moderate size of the project, all numbers are relatively volatile—ϕ_t meanders strongly around a general mean of 0.1 from time range 20 on.
- For Git, the number of *cochange* and *mail* edges does not follow the growth trend in the number of developers, rather, both values stagnate or decrease from time range 14 on. This is a sign of decreasing per-capita interaction with growing number of developers. Together with *mail* activity, ϕ_t slowly decreases from around 0.20 to around 0.15 over time.
- QEMU shows the opposite behavior of ownCloud: The mailing list that we retrieve for modeling the *mail* edges seems to be in usage only from time range 25 on, and then consistently with a ϕ_t that slightly decreases from around 0.30 to 0.25, as the *mail* activity does not grow as much as the *cochange* activity.

Our results suggest that, usually, only in early phases of the projects, there can be more *mail* activity than *cochange* activity. The correlation between the presence of *mail* and *cochange* edges is relatively stable for most projects, never exceeding values of 0.5 and usually fluctuating around 0.2 to 0.3. The relationship between *mail* and *cochange* edges is weakest for ownCloud and strongest for Jailhouse, which can be partially explained by the small number of active developers in Jailhouse. ownCloud turned out to be an exception as the mailing list is abandoned in later phases of the project, and we expect that this will influence the results of the decomposition and prediction that we will discuss in the next sections.

4.4.2 Decomposition Insights

Next, we seek to extend the insights from the descriptive statistics in Table 4.3 and the time-based investigations of the correlation between *mail* and *cochange* activities that we have discussed in the previous paragraph. We apply a canonical tensor decomposition and see whether it helps to understand the project dynamics even better by making group constellations visible. These group constellations are of exploratory nature and can (with our chosen decomposition methodology) not be tested statistically, but they may help to identify the sources and dynamics of project changes.

In particular, we computed the tensor decomposition for different numbers of groups R, ranging from 1 to 20. For the purpose of discussion, we had a look at the decomposition results for values of R between 2 and 9 (all of which are available on our supplementary website) and chose the ones that reveal interesting patterns to us. These patterns are therefore prone to overfitting. For description and exploration, large values of R can be helpful, whereas for prediction, the influence of R can be cross-validated with regard to predictive performance and usually is chosen smaller, as patterns in the past are usually not very indicative for the future.

For illustration, we discuss the canonical decomposition of the four-dimensional tensor $\underline{\mathbf{Z}}$ for BusyBox with $R=8$ (Figure 4.7). The decomposition results in estimates for $\lambda_{r=1}, \dots, \lambda_{r=8}$, $\mathbf{u}_{r=1}, \dots, \mathbf{u}_{r=8}$, $\mathbf{c}_{r=1}, \dots, \mathbf{c}_{r=8}$, and $\mathbf{d}_{r=1}, \dots, \mathbf{d}_{r=8}$, which are visualized in panels (a) to (d) of Figure 4.7, respectively. In what follows, we discuss some selected observations on this decomposition:

(a) Figure 4.7(a) shows the weights of the latent groups, $\lambda_1, \dots, \lambda_8$. While these values should not be interpreted in absolute terms, their relative size is important. The second (r=2) and eighth (r=8) factors have largest λ_r and therefore the strongest correlation with the probability of developers to interact. The sixth component is the least influential for the overall number of interaction events.

(b) Figure 4.7(b) shows which developer i is involved in which component r, u_{ir}. The plot shows a selection of the five most central developers[14] per component, which results in 15 developers for BusyBox as sometimes the top 5 are overlapping. The figure reveals aspects such as that developer 158 is relevant only for components $r=1$ and $r=4$. In the other components, this developer does not play an outstanding role. The most clear-cut group can be seen in component $r=6$, where the contrast between important and non-important developers is most pronounced. Developer 45 is a special person who seems to be involved in all the different phases and topics except component $r=6$.

14 Because of ethical considerations, we do not state the names of developers, but use pseudonymized developer ids.

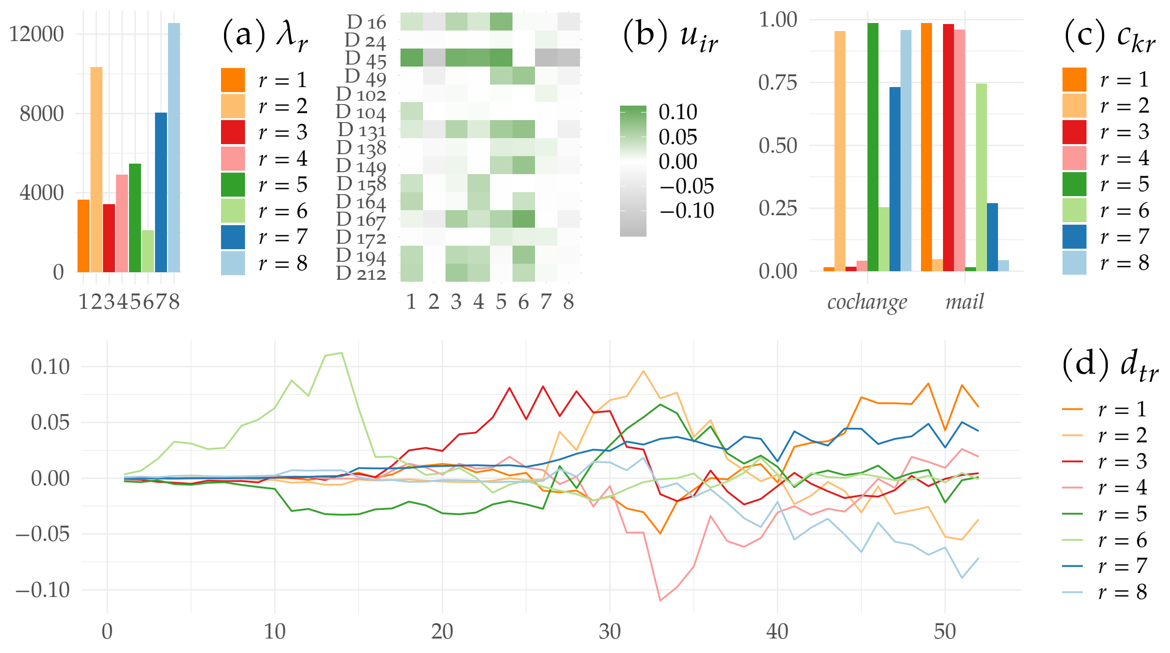


Figure 4.7: Results of the canonical tensor decomposition for BusyBox and $R = 8$. Panel (a) shows the weights of the factors λ_r. Panel (b) shows the developer effects u_{ir} for a selection of developers with the highest values in the first five factors. Panel (c) shows the interaction channel effects c_{kr}. Panel (d) shows the dynamic weights d_{tr} and thereby which factor was important at what time.

(c) Figure 4.7(c) shows which interaction channels are affected by the eight components (i.e., groups of developers). Components $r=1$, $r=3$, $r=4$, and $r=6$ describe predominantly *mail* activity. That is, the respective developer groups arise from joint e-mail activity. Their c_{Mr} is large in absolute value compared to their c_{Cr} value. The other four components describe mostly *cochange* activity (i.e., groups of developers connected by joint editing activity), with varying shares of *mail* activity.

(d) Figure 4.7(d) reveals that the component $r=6$ describes developer interaction activity in the early phase of the project, time ranges 10–15. Remembering the observations from Figure 4.6, we look more closely at the time period between $t = 14$ and $t = 16$: At this time, a sudden decrease in activity in component $r=6$ can be seen. This component relates predominantly to *mail* activity, as we learned from Figure 4.7(c). Most components cover a time period of 15 to 25 time ranges, that is, approximately 4 to 6 years.

Let us now synthesize the information from the different panels of Figure 4.7 and investigate the distortion between $t = 14$ and $t = 16$ more thoroughly: At time $t = 14$, the activity level of component $r=6$ has a peak, as we can see in Figure 4.7(d). Shortly before, component $r=5$ starts growing, with two new developers 16 and 45, without developers 194 and 212, as Figure 4.7(b) reveals. Figure 4.7(c) finally tells us that the new group is glued mainly by *cochange* activity whereas the "old" group had intensive exchange via the mailing list. After $t = 15$, a new mailing group arises ($r=3$), which describes *mail* exchange between developers 16, 45, 131, 167, 194, and 212. That is, mainly developers 49, 149, and 164 dropped out and developer 45 joined compared to component $r=6$.

While it is not advisable to prematurely draw conclusions from these insights, as the insights depend on R and on previous assumptions, they can still provide useful information for further investigation. Like for BusyBox, we visualized the tensor decompositions for different numbers of groups R between 2 and 9 for each subject project and screened the decomposition results for interesting findings about the developers and group constellations:

- Jailhouse: As expected, the strong association between *cochange* and *mail* leads to a poor differentiation between *mail*- and *cochange*-related groups. For $R = 3$, however, a specific *mail* pattern arises that excludes two of the developers that are usually involved in the *cochange* activities. This pattern can be used to investigate what the discussion in time ranges 7 and 8 was about, and why some of the core developers did not participate. As Jailhouse is a small project, spurious patterns are likely to arise with growing R, and we found signs for overfitting already for $R = 4$.
- OpenSSL: $R = 2$ and $R = 4$ reveal that, for the first 40 time ranges, the project has been dominated by developers 8, 13, 18, 119, and 134. Thereafter, new developers joined. $R = 4$ suggests that these new joiners were active on the mailing list, whereas *mail* activity excluded developer 18.
- ownCloud: As expected, *mail* activity plays a subordinate role. But at $R = 3$ already, the *mail* activity is separated in the component $r = 1$, and the time trend shows that this sub-group indeed phases out later. Also, developer 320 seems to have influenced the project mainly in the time ranges 15 to 25.
- QEMU: When going with $R = 5$, the two *mail*-related sub-groups $r = 3$ and $r = 4$ were dominated by the developers 75, 245, 594, and 621 (for $r = 3$) and 65, 89, and 115 (for $r = 4$). Also, the bulk of the discussions of the fourth group, $r = 4$, happened before the discussions of the third group, $r = 3$. It can also be seen that the social dynamics in the project have probably been most affected by the decrease in activity of developers 89 and 115, who were active mostly between time ranges 25 and 40. This is a valuable insight for investigating the stability of the project's group structure over time.
- Git: $R = 6$ reveals that the mailing list was heavily used from time range $t = 5$ on, and its activity level peaked at $t = 14$. While the social relationships in the mailing list remain constant, their importance declined over time.
- Wine: The decomposition for $R = 6$ reveals that, in early phases of the project, especially developers 38, 254, 301, 351, 619, 684, 691, and 833 were forming strong bonds, mostly performing *cochange* activity. When using larger R, more distinguishable sub-groups show up. $R = 9$, for example, reveals a particular communication component $r = 7$.
- Django: Here, *mail* and *cochange* networks are tightly integrated. Only for $R = 9$, a *mail* component ($r = 4$) reveals itself with a very clear-cut sub-group. In combination with the relatively small value for λ_4, it becomes clear why this component shows up only for large values of R: This component affects only few *mail* interaction edges.
- FFmpeg: There is no differentiation between *mail* and *cochange* activity for $R = 4$. The sub-group that involves a clear-cut subset of developers, $r = 4$, shows an activity peak between time ranges 18 and 25.
- U-Boot: At $R = 5$, a specific *mail* component can be distinguished from the other components. This communication sub-group gradually augments its activity from time range 45 on. The other sub-groups contain mainly *cochange* events. The interpretation of the *mail* patterns becomes more detailed at $R = 9$.

The results meet our expectations: In phases of strong correlation of *mail* and *cochange* activity, the modes of interaction do not split into separate components. Larger projects, which can be expected to have more complex social dynamics and structures, often profit from higher values of R. When interpreting the results and looking for interesting findings, it is worth trying different values for R. In our cases, $R = 3$ for small projects was usually enough. Also, when there was much more *cochange* activity, the *mail* patterns were partially hidden in lower-rank decompositions, whereas the higher-rank decompositions revealed more detailed social dynamics at the risk of overfitting.

4.4.3 Predictive Performance by R

As shown in Section 4.4.2, a large rank R of decomposition may reveal more detailed insights into group evolution in a project. In what follows, we also verify whether these insights are useful for prediction. As explained, if the sub-groups really describe long-term stable social relationships, they should be helpful for predicting future interaction. As mentioned in Section 4.2, a strong rank reduction can lead to overlaid patterns and reduced information content, hiding important information on dynamics. A high value for R, however, can lead to spurious patterns and interpretations. One way to assess whether patterns are spurious is to see whether they help in predicting the future. All results of this section (for all projects and for different values of R) can be checked on our supplementary website.

We start with a comparison of the predictive performance of our models for BusyBox over time. In Figures 4.8, 4.9 and 4.10, we show the predictive performance for BusyBox for $R = 2$, $R = 3$, and $R = 5$. Especially, before $t = 40$, that is, before BusyBox started into the growth phase identified in Figure 4.6, a large rank for the decomposition severely affects the prediction performance and lets them drop below the performance of the models naive and sum. A rank-2 reduction for the third-order tensor (i.e., model 3d) provides very good scores for predicting *mail* events right from time range 26 on. For $R = 2$, all reduced-rank models consistently outperform naive and sum. That is, a strong focus on the core developers helps in identifying future pairs of developers that will interact again. A differentiated perspective on the overlapping groups, however, appears to add more noise to the prediction and seems not to be very helpful.

Next, we validate the interplay of choice of R and $\overline{AUC}$ for BusyBox. Only the four models 3d, 4d, 3d-ext, and 4d-ext involve a rank reduction, so we visualize the changes in performance only for these models. In Figure 4.11, we show the predictive performance of the models for BusyBox by forecast horizon and the choice of the number of components R between 1 and 20.

While the forecast performance for $h=1$ seems to be invariant to the choice of R, a longer forecast horizon profits from a weaker rank reduction. That is, while the shorter forecast horizon profits from a stronger reduction of noise, the long-term prediction profits from the identification of more specific patterns. The optimum seems to be reached at around 10 components. This observation is relatively consistent for *cochange* and *mail* data, however, $R = 2$ performs better for predicting $h = 5$ than $R = 5$ for the *mail* data (independently of whether *mail* and *cochange* are analyzed simultaneously or separately—the identification of group structures and trends in the weights of these structures does not seem to help to predict *mail* interaction in BusyBox).

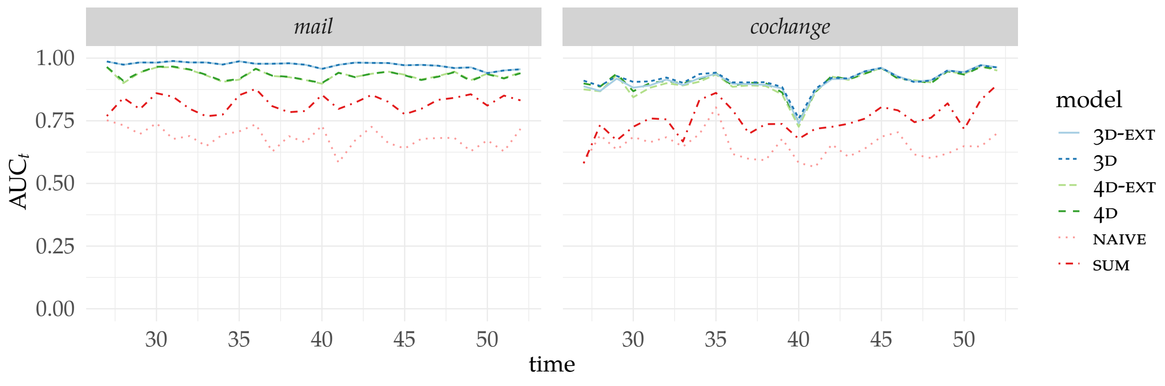


Figure 4.8: AUC$_t$ by type of interaction and time for BusyBox, $h = 1$, for $R = 2$.

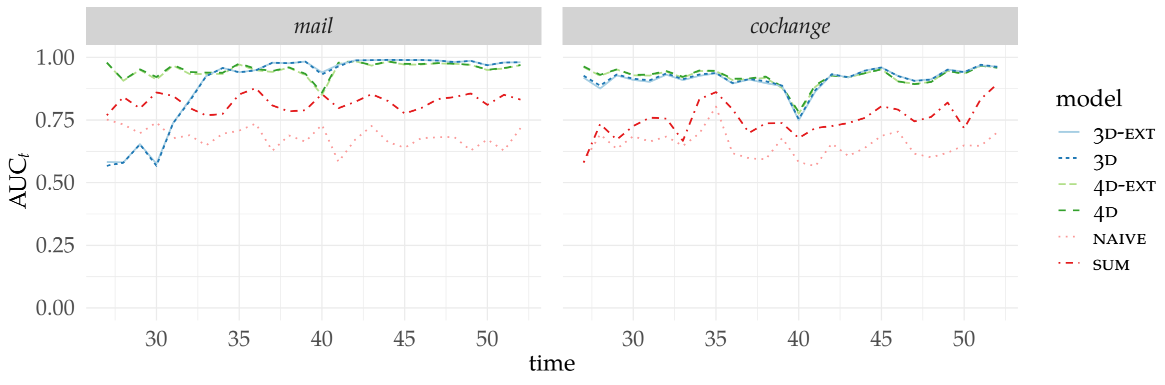


Figure 4.9: AUC$_t$ by type of interaction and time for BusyBox, $h = 1$, for $R = 3$.

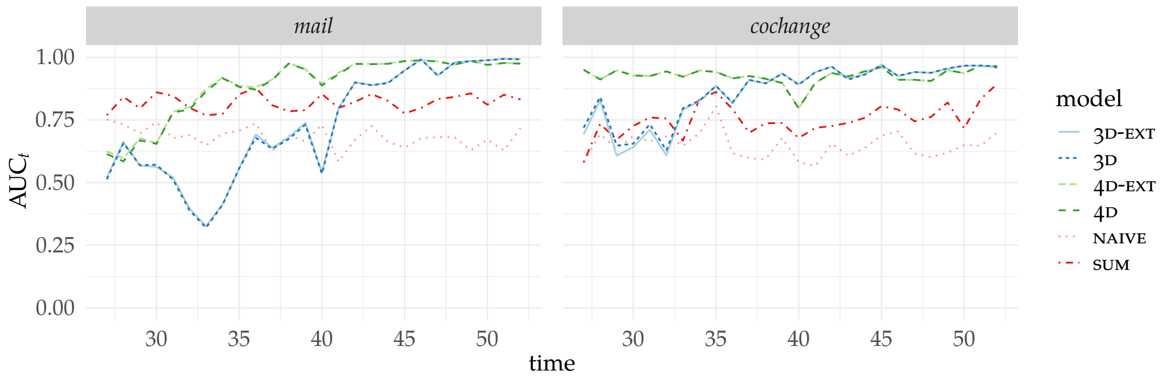


Figure 4.10: AUC$_t$ by type of interaction and time for BusyBox, $h = 1$, for $R = 5$.

For the other projects, the results are similar. For small R, 3D and 3D-EXT, as well as 4D and 4D-EXT are similar in performance, and differentiation only happens when R is increased. This speaks in favor of the exponential smoothing to reduce noise in the decomposition of quickly changing time trends. In general, the decomposition models' performance improves over time, while NAIVE and SUM perform similarly all the time. In some projects, such as Django, model 3D-EXT performs a lot better than the models without extrapolation for high R values.

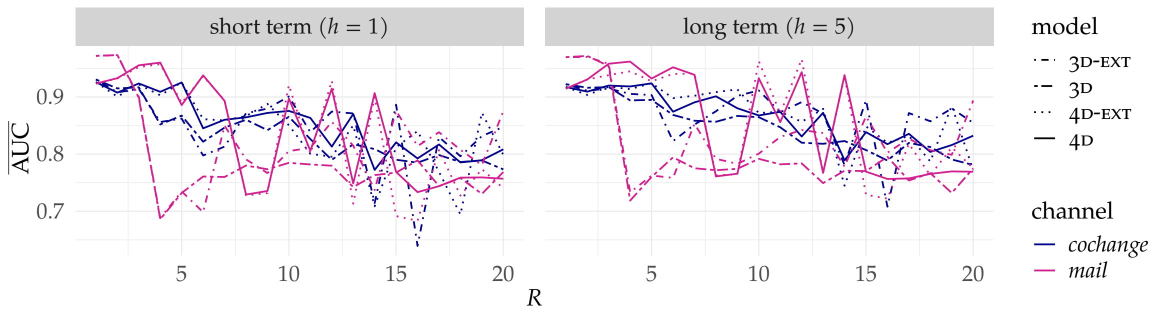


Figure 4.11: $\overline{\text{AUC}}$ by forecast horizon h and rank R of reduction for BusyBox. As expected, a small number of components is beneficial for prediction. The relative performance of model 4D decreases with higher R values, consistently for both forecast horizons and communication channels.

The extrapolation seems to smooth out some of the disadvantages of choosing higher R by smoothing the relatively volatile time weights.

The results imply that, while a rank reduction and the encompassing concentration on core developers is beneficial, the group patterns themselves change only slowly, and five latent factors are enough to describe the underlying structure of *mail* and *cochange* interaction. Small values for R usually result in good prediction results that outperform naïve predictions (models NAIVE and SUM). However, $R = 3$ can already lead to overfitting, and larger values of R lead to instability in prediction performance and are only suitable for description, not for extrapolating group structures to the future.

4.4.4 Overall Performance for Fixed R

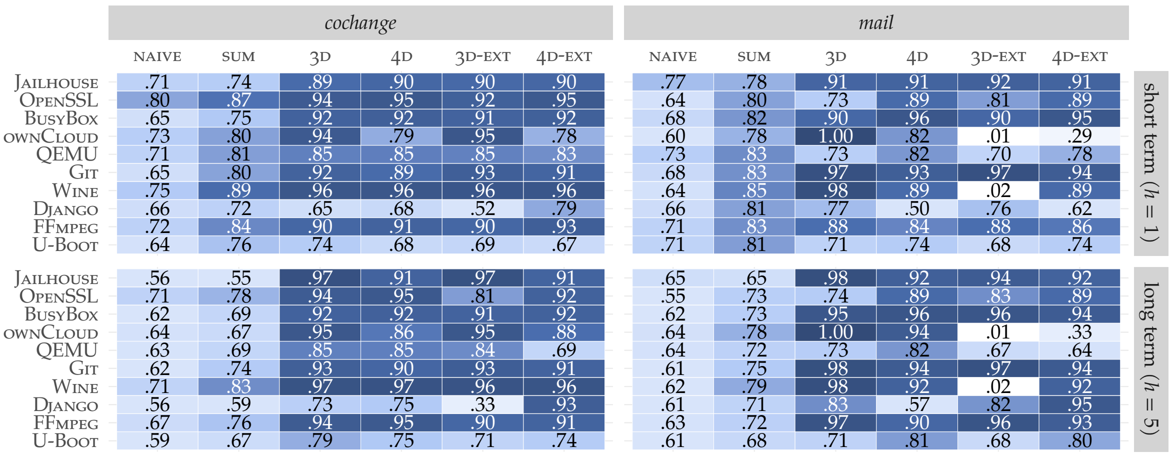
To avoid overfitting and discussions about R, we conduct the next analyses with a fixed $R = 3$. In Table 4.4, we show the average performance, first averaged over time by project, and then averaged over all projects (separate for two different forecasting horizons—1 year ahead or in the next time range). The results are also shown on our supplementary website.

In Figure 4.12, we show the overall performance averaged across time, for each subject project. The figure reveals that the averaged measures are highly influenced by a few outliers where the method makes no sense given the data quality. The three-dimensional decomposition with time-series extrapolation (model 3D-EXT) fails for the *mail* network for ownCloud and Wine. For ownCloud, this problem is due to the non-usage of the mailing list. This phenomenon is also responsible for the 100% accuracy that Figure 4.12 reveals for ownCloud (model 3D). In the case of the $\overline{AUC}$ values for method 3D for both forecast horizons, the numbers show that this instability is valid only for $R = 2$ and $R = 3$. For values larger or equal to 4, it performs similar to the other methods. Except for this instability, the results look like what we expected. Social relationships among developers seem to be least stable for project U-Boot, where edges are hardest to predict and the $\overline{AUC}$ values are smallest.

In general, model NAIVE performs worst. Table 4.4 shows that, for $h = 1$, $\overline{\text{AUC}}^{\text{NAIVE}}$ is, at least, 0.10 points lower than any other method for the prediction of *cochange* events, and, at least, 0.11 points lower than any other method for the prediction of *mail* events (neglecting

Table 4.4: Overview of the average performances of the different models. ∅ indicates the arithmetic mean of all $\overline{\text{AUC}}$ measures by model and forecast horizon $h = 1$ or $h = 5$, for $R = 3$. Model 3D performs best for all four combinations of interaction mode and forecast horizon.

	cochange						*mail*					
	NAIVE	SUM	3D	4D	3D-EXT	4D-EXT	NAIVE	SUM	3D	4D	3D-EXT	4D-EXT
∅ $(h = 1)$	.70	.80	.87	.85	.85	.86	.68	.81	.86	.83	.66	.79
∅ $(h = 5)$	.63	.70	.90	.88	.83	.88	.62	.73	.89	.87	.68	.83



	cochange						*mail*						
	NAIVE	SUM	3D	4D	3D-EXT	4D-EXT	NAIVE	SUM	3D	4D	3D-EXT	4D-EXT	
JAILHOUSE	.71	.74	.89	.90	.90	.90	.77	.78	.91	.91	.92	.91	short term ($h = 1$)
OPENSSL	.80	.87	.94	.95	.92	.95	.64	.80	.73	.89	.81	.89	
BUSYBOX	.65	.75	.92	.92	.91	.92	.68	.82	.90	.96	.90	.95	
OWNCLOUD	.73	.80	.94	.79	.95	.78	.60	.78	1.00	.82	.01	.29	
QEMU	.71	.81	.85	.85	.85	.83	.73	.83	.73	.82	.70	.78	
GIT	.65	.80	.92	.89	.93	.91	.68	.83	.97	.93	.97	.94	
WINE	.75	.89	.96	.96	.96	.96	.64	.85	.98	.89	.02	.89	
DJANGO	.66	.72	.65	.68	.52	.79	.66	.81	.77	.50	.76	.62	
FFMPEG	.72	.84	.90	.91	.90	.93	.71	.83	.88	.84	.88	.86	
U-BOOT	.64	.76	.74	.68	.69	.67	.71	.81	.71	.74	.68	.74	
JAILHOUSE	.56	.55	.97	.91	.97	.91	.65	.65	.98	.92	.94	.92	long term ($h = 5$)
OPENSSL	.71	.78	.94	.95	.81	.92	.55	.73	.74	.89	.83	.89	
BUSYBOX	.62	.69	.92	.92	.91	.92	.62	.73	.95	.96	.96	.94	
OWNCLOUD	.64	.67	.95	.86	.95	.88	.64	.78	1.00	.94	.01	.33	
QEMU	.63	.69	.85	.85	.84	.69	.64	.72	.73	.82	.67	.64	
GIT	.62	.74	.93	.90	.93	.91	.61	.75	.98	.94	.97	.94	
WINE	.71	.83	.97	.97	.96	.96	.62	.79	.98	.92	.02	.92	
DJANGO	.56	.59	.73	.75	.33	.93	.61	.71	.83	.57	.82	.95	
FFMPEG	.67	.76	.94	.95	.90	.91	.63	.72	.97	.90	.96	.93	
U-BOOT	.59	.67	.79	.75	.71	.74	.61	.68	.71	.81	.68	.80	

Figure 4.12: $\overline{\text{AUC}}$ measures by interaction channel, subject project, model, and forecast horizon. Projects are ordered by the involved number of developers (see Table 4.2) and organized in rows. Columns represent prediction methods, and the four main blocks represent interaction channel (horizontally) and forecast horizon h (vertically). $R = 3$ in all models that involve a rank reduction (3D, 4D, 3D-EXT, and 4D-EXT).

model 3D-EXT due to the above described problems regarding *mail* events in projects OWNCLOUD and WINE). The low performance of model NAIVE is also true for the long-term forecast, $h = 5$, where $\overline{\text{AUC}}^{\text{NAIVE}}$ leaks behind the other models. Model SUM achieves the second last position, but its performance is relatively close to the rank-reduction-based models 4D-EXT, 3D-EXT, and 4D. Model 3D-EXT performs bad for the prediction of *mail* events, but Figure 4.12 shows that this is due to the instabilities of the model in OWNCLOUD and WINE. As can be seen in Figure 4.12, for the models NAIVE and SUM, it is harder to predict interaction 1 year ahead than in the next 3 months: The $\overline{\text{AUC}}$ values are consistently smaller for the forecast horizon $h = 5$ than for $h = 1$. However, this does not hold for the tensor-decomposition-based models (3D, 4D, 3D-EXT, and 4D-EXT): Here, the $\overline{\text{AUC}}$ values for the forecast horizons $h = 5$ and $h = 1$ are relatively close to each other, in many cases even higher for $h = 5$.

In many cases, the models 3D and 3D-EXT perform similar to 4D and 4D-EXT, as can be seen in Figure 4.12. Averaging over all projects, 3D performs better than 4D, whereas 4D-EXT performs better than 3D-EXT (see Table 4.4). Hence, we are inconclusive whether sub-groups that exchange e-mails do also collaborate in programming tasks. We also do not see a clear difference between the weight extrapolation models (3D-EXT and 4D-EXT), which are robust to alliances of changing performance, and the non-dynamic prediction models (3D and 4D). This inconclusiveness might also be introduced by overfitting.

The intuition that there is low congruence of *mail* and *cochange* activity (an intuition derived from the $n_{11,t}$ measure) is confirmed by this analysis: Model 4D-EXT performs only slightly better than model 3D-EXT for *cochange* prediction, and is inconclusive for *mail* prediction. The consideration of information on one interaction channel is of little to no predictive power for predicting the other interaction channel. The performance of the reduced-rank decomposition *without* extrapolation (i.e., models 4D and 3D) is equal to or sometimes even better than the reduced-rank decomposition *with* state-space extrapolation (i.e., models 4D-EXT and 3D-EXT).

4.4.5 Answers to the Proposed Research Questions

When evaluating the predictive power of the different models, we can clearly see that all the models that are based on tensor decomposition perform better than the simpler baseline models. Hence, by using tensor decomposition and rank reduction, we can identify more stable, latent group structures in the communication and co-editing activity of OSS projects. This answers RQ_1:

Answer to RQ_1: With our method, we are able to identify stable group structures in OSS projects. That is, we identify groups of developers that steadily interact with each other during the project's evolution. Our results show that there can be distortions in the group structure (e.g., some developers joining the project take over the core work whereas, at the same time, previous core developers tend to restrict their activity to communication only).

The manifestation of these group structures varies in the different subject projects. In particular, co-editing group structures and communication group structures are project-specific and do not necessarily overlap. This answers RQ_2:

Answer to RQ_2: The overlap between the group structures that emerge from communication and the group structures that emerge from co-editing source code is project-specific: In some projects, there is a high overlap between co-editing and communication group structures (i.e., communication behavior and co-editing behavior of developers seem to be very similar), whereas in others, the overlap is small or even not recognizable.

Finally, regarding RQ_3, in which we ask whether considering past activities improves the prediction of future co-editing or communication activities, we cannot provide a single answer, as the predictive performances of the models that use tensor decomposition together with state-space time-series extrapolation perform differently on different projects when compared to the models that use tensor decomposition without state-space time-series extrapolation. Consequently, the emergence of group structures on different interaction channels is project-specific, and considering dynamic weights improves prediction of future activities in some of the investigated projects whereas it does not in other projects.

Answer to RQ_3: Considering past activity performs similarly well as without. In some projects, considering past activity slightly improves the predictive performance compared to not considering past activity. Some alliances seem to be of changing importance, whereas others seem to maintain a consistent importance. It also is project-specific whether the prediction on one channel can be improved by considering past activity on the other channel.

In summary, our method is capable of identifying stable group structures. Relying on past activities for prediction or expecting similar group structures for co-editing and communication activities sometimes leads to an improvement in prediction. This indicates that it cannot be assumed that the "mirroring hypothesis" is fulfilled, in general, as many project specific characteristics may influence the relationship between co-editing and communication. Our approach, though, can help to investigate questions of modularity and organizational patterns in several modes of interaction simultaneously. The tensor structure can easily be extended to three or four channels of interaction. Alternatively, interaction in different contexts can be one of the modes of the tensor.

4.5 Discussion

Our work has two main areas of contribution: methodology and empirical insights into developer behavior. We will start with a discussion of the strengths and weaknesses of our method and models in Section 4.5.1, and then continue with the discussion of our insights into developer behavior from a research perspective in Section 4.5.2 and from a practitioner's perspective in Section 4.5.3.

4.5.1 Methodology

We have transformed a multiplex network with event-driven edges into a tensor structure and applied a tensor decomposition as a clustering algorithm for visualizing group dynamics over time. We added a prediction model to quantify the predictive strength of different model variants. For analyzing the dynamics of group structures in socio-technical networks, we adapted the framework of Kunegis et al. [KFB10] to a multi-dimensional tensor structure. As this results in a relatively long vector of dynamic weights $\mathbf{d}_r$ for every latent component, we were able to combine the tensor decomposition with a state-space time-series framework for predicting future weights of the components, and thereby future interactions. Our exploratory and descriptive approach provides insights into sub-groups and temporary priorities in social networks with multiple types of interaction.

The method we have proposed has a few caveats that are due to its aim to simplify the understanding of processes and to make the dealing with large, multi-modal networks computationally feasible. The first noticeable simplification of our modeling approach is that event data (the creation of a commit or the response to an e-mail thread) is considered as a prolonged state of the network lending itself to a topological description. This simplification allows us to discretize time and to explore the current state of a network. We understand our approach as a heuristic to understand group dynamics in an exploratory way. Due to the lack of statistical hypothesis tests, most of our conclusions are built on the validation of predictive performance instead of testing. Another simplification is the usage of binary, unweighted edges. Tensor decompositions require relatively equal weight of every mode to bring insights into all interaction modes. In general, we found fewer mailing-list activities than co-editing activities. We could have standardized the single modes, but decided to use a binary edge, as it kept the weights of the modes up. When searching for groups in both modes, it can be adequate to use weighted edges and standardize by mode instead.

Our results show that the denoising of the networks via rank reduction, as well as the overlapping, dynamic group model describes the networks better than a simple assumption on repeated collaboration. For small values of R, a comparison with a rank reduction *without* state-space time series (model 4D) showed that this improvement is likely to be due to the rank reduction, not the state-space time-series model and exponential smoothing. For large values of R, the time-series method smoothed out the volatility that we first introduced with the many sub-groups that the method was allowed to look for. In combination, a large R can bring more detailed insights into group structures, and the extrapolation smooths out the volatility, and the combination of both a rank reduction and extrapolation brings more robustness.

In the following, we emphasize and demonstrate that our method is based on a modular concept of combining different statistical tools.

Modular Concept of Combining Statistical Tools Our method combines well-established statistical tools and methods that express our assumptions on the communication and co-editing behavior of developers. One of our core aims was high robustness and transferability of our method to contexts in which multiple types of interaction behavior need to be explored over time. Our method scales well to multiple types of interactions and also to large projects with many developers. To achieve this, we used tensor decomposition and prediction methods that make little assumptions on statistical distributions. At the same time, these methods are not suitable for testing statistical hypotheses (e.g., whether the grouping behavior of developers is a significant driver of developer behavior). Neither are these methods suitable for calculating prediction intervals (e.g., by validating whether observed group growth lies within the predicted range of values). We think of our method as a chain of modules that, if needed, can be replaced by different clustering and prediction algorithms.

In general, we base our approach on the assumption that it makes sense to represent communication and co-editing events in a time-slice tensor structure, as the observed events describe a temporal, enduring relationship. Other types of link prediction models such as exponential random graph models forgo this simplification and try to detect whether grouping behavior would, indeed, lead to networks that look like the observed network [LKR12]. These models do not detect clusters in a specific network but rather test hypotheses on the drivers of interaction behavior. Foundational work on this alternative logic was done by Quintane et al. [QCT+14]. Our approach, however, seeks to identify clusters in observed networks with multiple types of interaction, it does not seek to prove that clustering behavior is a generative pattern of the networks. Clustering is an unsupervised machine-learning technique. That is, it seeks to find a set of vertices in a network that are close to each other in the same cluster and far away from different clusters [CCZ+19]. In multiplex networks, common cluster (i.e., community or group) structures can emerge across the different types of interaction [DKK11; KBK05].

Under the assumption that the tensor representation has been accepted as a useful representation of developer networks, and that the groups we look for manifest in the temporal slices of the tensor structure, tensor-decomposition methods can be used for clustering. Chen et al. [CCZ+19] provide an overview of the most commonly used variants of tensor decomposition for clustering in multiplex networks. They also propose a clustering method that allows to estimate different groups across two modes of interaction. This approach would complicate the visual representation and interpretation of the decomposition outcomes, but could provide an alternative approach for model 3D to capture channel-specific group structures.

As we wanted to avoid any dependency between the identified groups in both channels in model 3D, we have chosen to cluster both interaction modes separately. The method proposed by Chen et al. [CCZ+19] differentiates between groups in both contexts, still using the entire tensor (instead of, as in our model 3D, decomposing two three-dimensional tensors to allow for separate group structures). As our method consists of a modular concept of combining statistical tools, the outcome of their decomposition approach would be compatible with the next steps of our method: There are cluster weights for a time range and stable groups over time that could be combined with a time-series prediction to extrapolate the current activity level of the groups and to predict future interaction.

As with the clustering step, also the prediction step can be easily changed to model additional assumptions on the development of group weights. For example, our current approach allows group weights to increase without bound. But—in real life—a developer's capacity and time are limited. Hence, an extrapolation method that includes a damping factor for growth could be more suitable. When using multivariate techniques (i.e., methods that model relationships between the single groups' weights and predict them simultaneously), one should keep in mind that the time series are actually related through their origin from a tensor decomposition. Therefore, we recommend to use univariate methods that do not involve assumptions on the statistical properties of the error term.

The above stated examples, on the one hand, relate central parts of our method to state-of-the-art work and, on the other hand, demonstrate the benefits of our overall method as a modular concept of combining statistical tools. In particular, these examples exhibit how certain parts of our modular concept can be replaced by other statistical tools and that our method could be easily applied and adjusted to additional assumptions in practice.

4.5.2 Perspectives for Empirical Software-Engineering Researchers

A key question of this work is how strong social ties and group structures in OSS projects are, and how congruent interaction via communication and co-editing is. We found substantial differences between the projects, which appear mainly related to the size and growth of a project.

The Identifiability and Stability of Group Structures The rank-reduction-based models (3D, 4D, 3D-EXT, and 4D-EXT) perform better than the non-reduction-based models (NAIVE and SUM). This confirms that ignoring occasional contributions helps in predicting future developer interaction and that there are core and peripheral developers. The canonical decomposition is able to identify groups of developers, where every developer can be member of several groups, and reduces the weight of peripheral developers (including developers who only interacted with others for a single time range). With that, our method determines the important developers based on their involvements in specific latent tasks as well as on the importance of these tasks. As our results indicate, this improves the prediction of future developer interactions. Therefore, our method can also be used to identify core developers of a project from a more nuanced view than state-of-the-art classification techniques do, which are merely based on standard network centrality measures and possibly ignore the latent group dynamics.

Interestingly, our results show that relatively long phases of stable weights d_{tr} arise in the canonical decomposition, which can be interpreted as evidence in favor of stable social relationships. These social relationships lead to repeated collaboration over a limited period.

That is, there are interactions among developers that persist for a specific period of time (e.g., while developers jointly work on a new feature), but as soon as these relationships may appear, that soon such relationships also can come to an end (e.g., the implementation of the new feature is finished). For instance, our decomposition insights for QEMU showed that a group of developers mostly participating in discussions on the mailing list for a time period of about 3.5 years mainly has affected the social dynamics of the project in this time period. At the beginning and end of this time period, structural changes in the project organization might have emerged due to the changes in the communication activity of these developers. Hence, this could be used as a starting point for analyzing the causes and effects of structural changes (perhaps driven by specific organizational events) on project success. So, this could be helpful for better understanding social dynamics and improving a project's software-development process.

The Correlation Between Communication and Co-Editing Our results also show that communication and co-editing have a consistent positive correlation (positive values ϕ_t between 0.15 and 0.21, omitting one outlier in each direction). This is, to some extent, in line with "Conway's Law" [Con68] or the "mirroring hypothesis" [CB16], in which it is assumed that the social structure (i.e., communication) of a software project mirrors its technical structure (i.e., co-editing) because of the coordination requirements that arise from co-editing software modules [CHC08; CWH+06; MC90]. This lets us assume that there is a (weak) socio-technical congruence, as the occurrences of communication and co-editing activities are positively correlated. Researchers have argued that the state of socio-technical congruence arises from the decomposition of programming activity into work items or tasks [KCD12; Par72]. In our setting—we are even more abstract—such work items can also be the latent factors r which we treat as unobserved tasks. When choosing larger values of R, usually the decomposition identified one or two specific sub-groups in communication. That is, most of the identified sub-groups are mainly related to co-editing, but with a high enough rank of decomposition, at least, one communication-related sub-group appears in our subject projects. Thus, mainly communication-related sub-groups are harder to detect. This could be an indication that there are only few pure communication-related sub-groups and most of the co-editing-related sub-groups also cover the corresponding communication part. Nevertheless, we need further investigations to find out whether the developers that are mostly involved in the communication-related sub-groups do also appear to be important in one of the co-editing-related sub-groups, and, if not, what this means with respect to socio-technical congruence.

Our results and our method can inspire research on software development with regard to different aspects. One important stream of research is to understand what developers usually discuss about, and how their communication is related to joint programming activities. Our study contributes to this field of research by showing that communication (at least, in terms of e-mail exchange on public mailing lists) and co-editing are only weakly simultaneously related. This result is in line with our results from Chapter 3, in which we have analyzed the synchronicity of cochange events and e-mail activity on the mailing list. Moreover, this result is also supported by the preceding study on synchronous development of Xuan and Filkov [XF14]. However, they identified a stronger correlation than we do, which might be specific to the chosen subject projects. The difference in the correlation between co-editing and communication activity might also be caused by the different operationalizations: They used simple time-series event data of 6 OSS projects, whereas we construct developer networks from a richer history of 10 OSS projects and apply a tensor decomposition on it with the aim of being more accurate in identifying latent structures. There might be multiple reasons why socio-technical congruence seems to be rather weak in the projects we analyzed (e.g., related co-editing and communication activities could be shifted such that they do not take place temporally close-by, or developers who mainly coordinate may not show up in co-editing activities any more, etc.). This observation has also been discussed in the literature [e.g., GBL+13; GKD+19; LC03] and needs further attention in future work.

Evolutionary Patterns in Open-Source Software Projects Research in empirical software engineering can profit from our method and models for finding patterns in the relationship between developers, without the need to explicitly specify all mechanisms and reasons that lead to the formation of modules, or hierarchy. Joblin et al. [JAM17], for example, investigated the emergence of scale-freeness, modularity, and hierarchy in OSS developer networks. While they used descriptive measures for showing these network characteristics, their approach could be combined with our probabilistic model of seeing the mentioned characteristics as latent phenomena that drive the emergence of edges in the networks. This could help to better understand the evolution of software projects and to identify changes in (latent) network characteristics, which might have influenced the organizational structure and coordination processes of an OSS project. Such a better understanding could be used to prematurely identify potentially harmful changes in the organizational structure of OSS projects and to be able to launch potential countermeasures for retaining project success. Especially modularity has a direct interpretation in terms of latent factors, and understanding and measuring modularity can provide important information to decision taking in OSS development: Modularity helps to keep coordination needs among developers reasonable, and often arises naturally due to common capabilities and interest in a common problem or challenge [JAM17].

While describing the topology of a network, factorization provides insights into the generation process, by carving out the latent patterns that have led to the observed structure. Latent factors, extracted for example via spectral decomposition, inherently represent clustering, local heterogeneity, and other characteristics such as maximum distance across the network, bottlenecks, and degree of randomness [Hof09; SR03]. Knowing about such latent factors, researchers can get new insights into growth processes in OSS projects and draw new conclusions regarding whether and how such characteristics of growth processes can influence project success. For example, as we have seen in the decomposition insights of BusyBox, the latent information regarding dynamic weights of sub-groups can be used to identify which latent factors (and, therefore, which sub-groups) are responsible for the temporary sharp drop of communication activity and to identify which developers are related to these sub-groups. In addition, the decomposition insights can also reveal which latent factors (i.e., sub-groups and developers) are responsible for the anew increase of communication activity after the sharp drop (e.g., whether these sub-groups are the same as before or whether other sub-groups have gained importance). Such investigations cannot only be undertaken to examine activity patterns, but also to investigate network distortions or formation processes, etc. [LHH17]. Besides, finding latent factors can also be used for community detection: As a latent factor describes a certain group of developers that work on a joint unobserved task, all the latent factors already represent overlapping groups of developers. However, the tensor decomposition does not only show us the groups but also provides us with further information on group importance, kind of involved activity (co-editing vs. communication), and dynamic importance over the complete project time span. Such information can provide more insights into the organizational structure of OSS projects than detecting groups based on simple network classification metrics. For example, in ownCloud, we can see that, when decomposing the tensor into three latent factors, there is a high overlap among the three developer groups (showing that many of the core developers have a stable position in the project structure over time). However, we can also see that a specific developer is only important for one of the three groups that only played a central role in a rather early project stage. This way, the decomposition provides us with more information about the groups, which could be used to examine the reasons why specific groups gain or loose importance.

In summary, our method can be used by researchers as an easy means to investigate project evolution. Later on, this can be helpful for project maintainers to get a deeper knowledge on the ongoing, past, and potential future activities and on the importance of specific developer groups that have emerged and vanished during project history. This information could be exploited to reveal coordination needs and improve the coordination of developers' activities with respect to specific organizational events (e.g., to coordinate knowledge sharing when a central long-term developer of the project is about to leave).

4.5.3 Perspectives for Practitioners

Our study also provides potential insights for practitioners: The information on the role of specific groups, on coordination needs, and on knowledge sharing could be collected and visualized in a dashboard, which could help developers to improve their coordination based on the knowledge about group structures and their importances. For instance, (potentially unexpected) instabilities or irregularities in the group structures of collaborating developers could be detected by our method. Project managers could then use the decomposition insights to identify sources of problems and develop concrete ideas on how to mitigate or even solve the problems identified via irregularities in the group structures. For example, in BusyBox, we recognized irregularities when new developers joined the project, when previous core developers left the project, and when the remaining group of previous core developers discontinued their co-editing activities but started concentrating on communication tasks only. These irregularities can be detected when having a closer look at our decomposition results (see Figure 4.7): The dynamic weight of the latent component that describes the main tasks of a group of previous core developers had a sudden decrease, while the dynamic weight of another component (which also contains some of these previous core developers, but only consists of communication activity) suddenly started to increase. At around the same time, a new group of developers (including two recently joined developers) emerges and takes over most of the co-editing activity. However, this new group has almost no communication activity, as the interaction-channel effects of the corresponding latent component reveal. Such an irregularity could be further investigated to uncover whether there is a potential lack of coordination. Knowing about such a collaboration issue could be beneficial for successfully integrating new developers (e.g., project managers could detect such issues and discuss them with the developers early).

4.6 Threats to Validity

The above presented results of our study may be threatened in many regards, as always in empirical studies. In the following, we discuss the potential threats to validity from two different perspectives: We first discuss methodological threats to validity. Thereafter, we elaborate on the potential threats to the validity of the empirical data that we have used in our study to evaluate our proposed models.

Methodological Threats to Validity (Construct Validity) One threat to the validity of our findings is the specification of the length of the time windows for our method. We chose 3-months time windows and an aggregated approach as we seek to describe the evolution of a topological pattern of the network: developer grouping or vertex clustering. The analysis can neglect the importance of people that replied to some messages only after the switch from one time range to another. The naïve baseline (NAIVE) is most vulnerable with regard to the time-window length, as it contains ever less information per time range when the time-window length decreases. The tensor decomposition is less vulnerable, as it always looks at the entire tensor. However, the time weights become more noisy, with more ups and downs in $\mathbf{d}_r$. Experimental work by González-Bailón et al. [GWR+14] illustrates the biases that arise when sampling network data: As we sample repeatedly and keep the edges that cross ranges, we have this risk of interpreting common changes as an interaction where there is actually no collaboration. For example, in our setting, two events can count as an interaction if the person who edited the file first has already left the project. This might be problematic for interpreting the identified group structures and using them for link prediction as we assume that this person was active all the time (and, therefore, had lots of interactions with various developers) whereas it never was active (except for the beginning). When discretizing interactions between developers into time ranges, in general, we can either lose interactions that go beyond the ranges of a time window or we can keep edges that go beyond time ranges and take the risk that some of these edges actually do not constitute a real interaction due to the potentially long time span between the two corresponding events, which leads to a distorted picture on the relationship between developers' activities. Torn between these two poles, we decided to keep interactions that go beyond time ranges, not to lose potentially important edges between developers.

Second, we have chosen a relatively simple decomposition method, which infers latent factors without indication of standard errors, probabilities, or similar quality criteria that would help assessing the fit of the decomposition and the group structures. The probabilistic model discussed in Anandkumar et al. [AGH+14a] describes in more detail how tensor decomposition can be used to infer group structures on the basis of the assumption of a Dirichlet distribution of the observed interactions. The method relates community detection to the mixed-membership community model of Airoldi et al. [ABF+08]. Much like their model, our models assume stochastic independence of single edges given the group members of the vertices in the network. However, to keep our calculations computationally feasible also for large projects, we decided to use a non-probabilistic decomposition method, to the disadvantage of hypothesis testing and precision estimation. When applying our method as a prediction method, the cross-validation procedure has to be adapted. We did not hold back a test sample to test the best R or model. This was not necessary because we wanted to describe the fit of the models for the given dataset: Which of the models with their basic assumptions best fits the observed data? Usually, sparser models perform better for prediction, and, therefore, we expect lower numbers of R to perform even better in a setting where the rules of holding back a part of the sample are being followed. Within the tool set of exploratory clustering of vertices, there may be better approaches. For example, including a penalty for the simultaneous membership in more than one group may help to achieve better results.

A third point worth discussing is the assumption of constant group patterns that vary over time only in their relative importance. Much like the models of Airoldi et al. [ABF+08] and Anandkumar et al. [AGH+14a], our models assume vertices to be members of different overlapping groups that can be described statically in strength of inter-personal relationships. We use the tensor decomposition to find patterns that describe a stable group behavior—groups that are active for a certain span of time and that collaborate via joint programming and communication activities. Only the relative importances of the groups change, through their λ_r weights. A different approach is to model the change in some few central groups via a clustering that results in one developer being member of only one group, but changing groups over time. This alternative result could also provide a parameter that estimates the speed of group-structure adaptation, measuring the changes from one time range to the next. We found our constant group understanding easier to use for a description of the entire project history—when a current state of the network is all that is needed, the other approach may be more helpful.

In future work, our analyses could be easily extended by modeling directed and weighted edges. For example, it is known who contributes to a mailing-list thread first, and who edits a source-code artifact first [JMA+15]. The binary values can be replaced by weights, such as the number of contributions within time range t, the number of code lines edited in the *cochange* events, or the number of responses of developers i and j on a mailing list.

Threats to Data Validity (Internal and External Validity) We used data extracted from 10 well-established OSS projects from different domains to validate our models empirically. Nonetheless, the validity of our data is threatened by several aspects.

As we identify developers by their names and e-mail addresses, one potential threat is that developers may use different spellings of their names and use various e-mail addresses, which makes it hard to impossible to precisely track the co-editing and communication events for each developer correctly. However, we use the heuristic for name and e-mail disambiguation from Oliva et al. [OSO+12], which matches developers by their names and e-mail addresses and which has turned out to work well in empirical studies [WSS+16].

Another potential threat is our general assumption that edges between two vertices in the network describe the co-editing or communication of these developers. This does not threaten our methodological outcomes, but is a potential risk to the interpretation of our empirical results, as two developers who reply to the same mailing-list thread or work on the same file might not necessarily work on a common task and might also not be aware of each other and each other's actions. However, many related studies in the literature also rely on this assumption and base their analysis of developer networks on this operationalization [e.g., BGD+06; BPD+08; Bir11; GHJ98; HSA20; JAM17; JMA+15; LRG+06], which also has previously been validated via developer surveys to be accurate and meaningful [JAH+17]. Nonetheless, even if there is no explicit common task and the developers do not really interact with each other, the edges between them in the network describe some latent interaction. Hence, such edges represent implicit interactions of developers and their contributions, arranging them in groups of developers working on similar tasks, which still is an important facet when investigating the evolution of a software project and its group dynamics.

To increase the external validity of our results, more subject projects have to be considered, and also other forms of communication, such as communication via chat or GitHub issues. Communication on a project's mailing list is not the only communication channel, and group patterns may also extend to other interaction channels. Nevertheless, we evaluated the performance of our models on 10 well-established OSS projects from different domains to provide some initial insights in how our models perform when looking at two different interaction channels, which already allows us to synthesize some implications on the performance of our models as well as for empirical software-engineering researchers.

4.7 Conclusion

OSS projects have a reputation for being anarchic compared to commercial software development. Some assume that this is one of their major strengths as it fosters for flexible organizational structures and integration of peripheral developers, who contribute a significant share of the value of OSS projects. Nevertheless, researchers found that repeated collaboration and stable social relationships are essential drivers of the quality and success of software development. In this chapter, we asked how strong social relationships in OSS projects are and how congruent sub-groups in communication and programming are. To provide an answer to these questions, we developed a modular, multi-step method to detect developer groups. We departed from a model that assumes groups in OSS projects to be overlapping, both across tasks and in time, and operationalized this model with a tensor data structure, a decomposition to find the groups, and a time-series extrapolation to predict the future behavior of the groups.

Furthermore, we proposed a probabilistic model to analyze the strength and dynamics of group behavior in OSS development. We model the groups as latent factor structures in a low-rank factorial structure, and assume that the social structure of a project contains R sub-groups. As any developer can work on multiple tasks, there is no restriction of one developer belonging only to one specific group. Instead, groups are overlapping. One latent factor consists of (1) developer effects that reflect the involvement of developer i in one particular group r, (2) interaction-channel effects that reflect how much this group involves either co-editing or communication ($k \in \{1,2\}$), and (3) time effects that contain the current weight of the sub-group for a particular time range t. This way of thinking and modeling is similar to collaborative filtering approaches, where correlations and similarities are mined without strong hypotheses on the nature and properties of the underlying group-forming mechanisms. We infer the latent factors from the observed data with the help of canonical tensor decomposition. To do so, we discretized the event data into T non-overlapping time ranges. Finally, we combined the decomposition with a state-space time-series model and exponential smoothing to extrapolate a trend and a current level of the relative importance of the single latent factors to future time ranges.

To assess the appropriateness and predictive performance of our models, we defined a series of baseline models. Baseline NAIVE assumes that everyone who co-edits or communicates at time T will also do so at time $T + h$. Baseline SUM sums up all the past interaction behaviors, putting more emphasis on older information and repeated interaction than NAIVE. Model 3D looks at communication and co-editing information separately, using third-order tensor decomposition, which resembles the state-of-the-art approach of analyzing the social and technical dimensions separately. Model 4D uses fourth-order tensor decomposition and simultaneously models groups in co-editing and communication. Both models 4D and 3D assume that the current relative weights of the patterns will be constant in the future. Finally, to account for changing group weights, in the models 4D-EXT and 3D-EXT, we use a time-series method to extrapolate the current group-weight growth or shrinkage to the future. The models 4D-EXT and 3D-EXT outperform the baseline models in the prediction of future edges. However, the extrapolation leads to instabilities when compared to the remaining approaches. Consequently, the models 3D and 4D turned out to be more robust to abnormalities in communication usage, such as the decreasing use or abandonment of mailing lists, and had a slightly better mean performance for prediction. While rank reduction and the incumbent concentration on the more central developers indeed benefit predictions, the state-space time-series model and trend extrapolation did not significantly improve predictive performance.

Our research shows that there is evidence that rank reduction is able to identify valid group structures and correctly describes the social structure of OSS projects. Our method is a relatively intuitive and exploratory way to investigate group structures in social and socio-technical networks, as it consists of a modular concept of combining different statistical tools. This way, our method can be easily adjusted to investigate various assumptions on developer behavior in OSS projects. Consequently, it can be used to better understand group dynamics in a computationally feasible way, even in huge networks with more than two types of interaction.

In empirical software-engineering research, our method can be used to explore the group structure in more detail and to understand collaboration structure, role diversification, and the usage of different modes of interaction. For example, we found that shortly before a temporary decrease in the communication activity of project BusyBox, there is a peak in the activity of a sub-group which had extensive e-mail exchange via the mailing list, whereas thereafter a sub-group mainly driven by co-editing activities started to be active. After the temporary decrease of the communication activity, a new sub-group arises that takes over only communication activity. However, recently joined developers (who are part of the sub-group that mainly performs co-editing activities) are not involved in the communication activity, which could point to a potential lack of coordination. Using our method, we are able to identify such distortions in the network structure and to easily analyze the underlying changes with respect to interaction channel, dynamic importance, and developer importance. We have investigated the group dynamics of co-editing and mailing-list communication in 10 well-established and long-running OSS projects and, via canonical tensor decomposition, we identified stable group structures (consisting of groups being stable for several years) as well as periods of sub-group instability in all the investigated projects. Some of these sub-groups are mainly related to programming activity, whereas others are mainly related to communication, and some are taking over the coordination activity (showing both communication and programming activities). While some sub-groups are persisting since the beginning of the project, others emerge in the course of project growth. Interestingly, project growth in terms of increasing programming activity is usually not complemented by increasing communication activity, as we detected only a weak correlation between co-editing and communication events. This phenomenon shows that there might be a need to improve developer coordination. Our method can help to detect patterns of instability in specific sub-groups (such as the abandonment or joining of important developers) by considering multiple interaction channels all at once, and to assess their effects on project success and potential coordination needs within or among specific sub-groups. Such insights can drive the improvement of virtual coordination platforms like GitHub by helping to determine the collaborators for a specific task or discussion (e.g., recommending project contributors who may be able to answer a question or should be informed about some specific aspect).

Hierarchical Organizational Structures in Open-Source Software Projects 5

This chapter shares material with Joblin et al. [*JEB+23*].

Many software systems are developed either as open-source software (OSS) or rely on OSS libraries, frameworks, etc. [Wei18]. OSS developers actively contribute to an OSS project by means of coding and by issuing and discussing bug reports, feature requests, etc. OSS developers typically organize in a decentralized and self-organized manner [BPD+08; CLW+07; Yu08]. Despite the absence of a formal process and a central command-and-control structure, prior studies have shown that developer organization is far from being a purely random process. Studies of developer organization based on network models have been accumulating growing evidence that multiple organizing principles are simultaneously at play [BPD+08; JAH+17; JAM17; JMA+15]. There are two important organizing principles: (1) The probability that a randomly selected developer has k connections to other developers in the network is described by a power law—the *scale-freeness principle*—and (2) developers organize into densely connected groups—the *community principle*. It has been shown that, to simultaneously reconcile these two principles under one roof, the groups must arrange according to a hierarchy (see Figure 5.1) [RB03]. Hierarchical organization induces a dependence between the number of connections to a developer (vertex degree) and the density of local connections (clustering coefficient), which can be used to test for the presence of hierarchy [RB03].

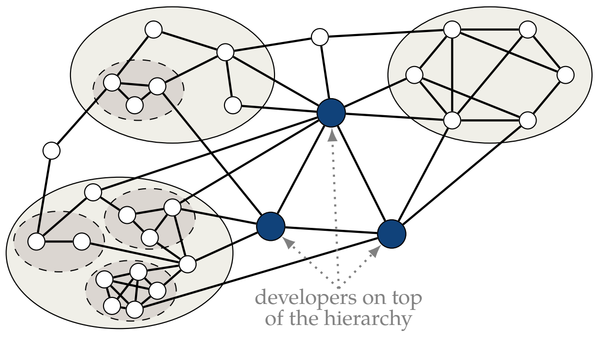


Figure 5.1: Even without formal process or mandated developer roles, a hierarchical structure emerges in OSS projects, with few developers on top of the hierarchy (blue, large vertices) and many developers lower in or even outside the hierarchy (white, small vertices) [JAH+17].

Recently, Joblin et al. [JAM17] found indications that, over time, highly successful OSS projects develop a *hybrid* organizational structure that comprises a *hierarchical* part and a *non-hierarchical* part. This observation is consistent with previous findings reported in the literature [CH06; MFH02] and is intriguing for two reasons: (1) it implies that hierarchical organization is not necessarily a global organizing principle and that a fundamentally different principle is at play below the lowest positions in the hierarchy; (2) the vast majority of developers often occupy non-hierarchically organized positions in the developer network. In this vein, previous work suggests a connection between role stratification and the emergence of organizational structure [JAM17]. *Role stratification* is the process of the emergence of developer roles arising from differences in the developers' behavior. Role stratification limits coordination overhead and improves information flow, this way, influencing project quality and performance [HR09; JAM17; LS07; OF07; TLV13; TPK19]. In particular, Joblin et al. [JAH+17] provide evidence that developer roles and hierarchy in developer networks are connected such that *core developers occupy upper positions* in the hierarchy and *peripheral developers occupy lower positions* [JAH+17], where *core developers* play an essential role in developing the system architecture, programming, and forming the general leadership structure, with long-term involvement, and *peripheral* developers help with bug fixes or small enhancements, with irregular or short-term involvement.

Given the influential role of developers in the hierarchical part, it is important to understand the interactions between the hierarchical part and the non-hierarchical part. To this end, we want to learn whether the *hybrid structure* is universal and how it evolves with project life cycles. In addition, we adopt the perspective of individual developers and investigate how they traverse through the organizational structure from on-boarding to off-boarding, how their support contacts (i.e., co-developers) are structurally distributed within the organization, and how tenure and programming activity relate to their structural position. An improved understanding of these aspects is important for two main reasons: (1) it provides insight into the organizational mechanisms that large successful OSS projects use to manage coordination and communication, which is ultimately useful for any large-scale, globally distributed software development project; (2) our insights lay the foundation to derive measures that encourage a project toward known successful organizational structures to increase the likelihood of success. On the one hand, hierarchy has certain functional advantages when it comes to efficiency, but the lack of information-channel redundancy makes it vulnerable in volatile conditions (e.g., high developer turnover). On the other hand, non-hierarchical structures with lots of built-in redundancy tend to be robust to volatile conditions, but are less efficient. By better understanding the dynamics and relationships between parts of the project that are organized differently, we can begin to understand which organizational structures are ideal given the behavior of members or groups and how they interact with each other. For example, it is plausible that newly on-boarded developers (which are likely more volatile) would not be ideal candidates to be positioned higher in a hierarchy, but rather begin outside the hierarchy and then become members of the hierarchical part over a period of time once it is clear they are a consistent contributor.

To address these questions, we conduct a longitudinal study on 20 popular OSS projects of various application domains and sizes with a total of 831 6-months snapshots. We explore their organizational structure as a network of developers who are connected by communication ties [LRG04]. Based on these developer networks, we divide the set of developers into a hierarchical part and a non-hierarchical part and explore their evolutionary trends (RQ_1), we track the neighborhood of individual developers to understand their placement and progression in the hierarchy (RQ_2, RQ_3), and we characterize the roles of developers in the hierarchy with information on tenure and activity (RQ_4).

In our study, we make a number of notable observations, including that (1) hierarchy is a pervasive structural feature of developer networks of OSS projects, (2) OSS projects tend to form hybrid organizational structures, consisting of a hierarchical and a non-hierarchical part, and (3) the positional trajectory of a developer is to start loosely connected in the non-hierarchical part and then tightly integrate into the hierarchical part, which is associated with acquisition of experience (tenure), in addition to coordination and coding activities. Furthermore, our study (a) provides a methodological basis for further investigations of hierarchy formation, (b) suggests a number of hypotheses on prevalent organizational patterns and trends in OSS projects to be addressed in further work, and (c) may ultimately help to guide the governance of organizational structures.

In summary, we make the following contributions:

- Insights regarding the presence and evolution of hierarchical and hybrid organizational structures in OSS projects and a method to identify these structures.
- Findings regarding the nature of the relationship between members of the hierarchical and non-hierarchical parts.
- A comparison of the developers in the hierarchical and non-hierarchical parts regarding tenure and activity level.
- A discussion of practical implications and hypotheses that shall guide further research.
- A replication package including analysis scripts, pseudonymized raw data, and results on a supplementary website: `https://hierarchypaper.bitbucket.io/public/` and `https://zenodo.org/record/7199267`.

5.1 Background & Related Work

In this section, we provide background information on different forms of higher-order network structure and related work that studied organizational aspects of software projects using developer networks.

5.1.1 Higher-Order Structure in Networks

Depending on the underlying organizational principles that influence the formation of edges in a network, the types of structural features the networks possess can differ significantly. For example, if the existence of each edge is determined purely by chance (i.e., by flipping an unbiased coin where heads corresponds to an edge and tails does not) then an Erdős-Rényi (ER) random network is generated as shown in Figure 5.2 (left) [ER59]. Due to the independent formation of each edge, these networks lack higher-order structure (e.g., communities or hierarchy). To achieve a departure from these purely random network structures, an underlying organizing principle must induce a dependence between the edges. For example, if groups of vertices exist such that forming edges among members of the same group is more likely than forming edges with members of different groups, then higher-order structure in the form of

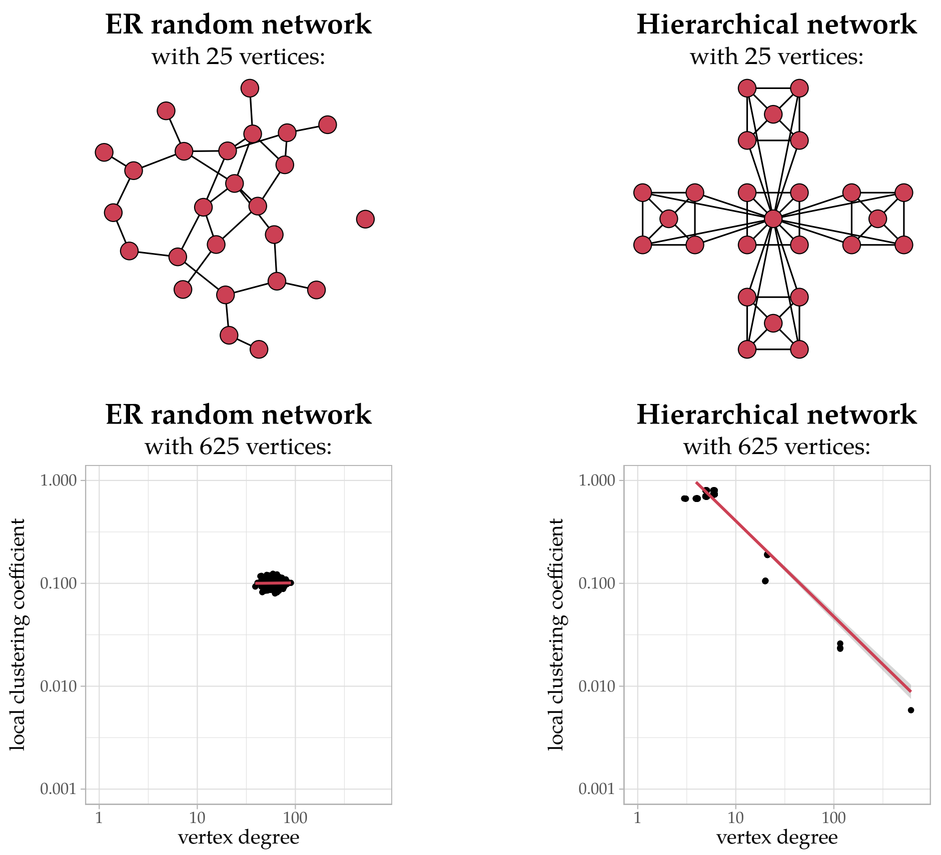


Figure 5.2: Comparison of Erdős-Rényi (ER) random networks (left) and hierarchical networks (right). Upper part: Visualizations of an exemplary ER random network and an exemplary hierarchical network with only 25 vertices each (for the sake of visualization). Lower part: Scatter plots that reveal the relationship between vertex degree and local clustering coefficient on a logarithmic scale in an exemplary ER random network and in an exemplary hierarchical network with 625 vertices each. The ER random networks are constructed using an edge probability of 0.1 for a pair of vertices, and the hierarchical networks are constructed according to the corresponding algorithm of Ravasz and Barabási [RB03]. This figure is based on Joblin [Job17, p.44] and inspired by Barabási [Bar16, pp.336–337].

communities arises [GN02]. Some real-world networks are also known to be scale-free, which implies that the degree distribution of vertices follows a power-law distribution [New05]. One way this property occurs is through a dependence between degree and the probability of edge formation, such that vertices with a higher degree are more likely to gain new edges than a vertex with a lower degree, which is known as preferential attachment [New05]. A third example of non-random structure is hierarchy. Hierarchical structure in networks can be achieved by inducing a dependence between the local clustering coefficient and the degree of the vertices in the network [RB03], as introduced in Section 2.3.3. Network hierarchy manifests as local clustering within a global tree-like arrangement of these clusters, which is shown in Figure 5.2 (right). Due to the stochastic nature of complex networks, the concept of hierarchy differs slightly from a strict hierarchy which forbids relationships between entities at the same layer. Moreover, hierarchy is distinct from community and core/periphery structures which do not require a dependence between the clustering coefficient and degree [RB03].

5.1.2 Using Developer Networks for Studying Organizational Aspects

For studying socio-technical aspects of software development, networks are used to represent relationships between developers induced by their development activities. Typically, networks are constructed by considering information from the version-control system (VCS) by extracting relationships from the commit activity [e.g., CHC08; CMR+09; CPT+19; JAH+17; JAM17; JMA+15; MW11; NMB08], from communication activity through issue trackers or mailing lists [e.g., BPD+08; HMR06; WSD+09], or a combination of both [e.g., BNG+09; MJT+22].

Network representations of software projects have proven to be a powerful abstraction in numerous applications. Multiple studies have demonstrated that developer networks can successfully predict software defects [BNG+09; MWS+08] and build failures [WSD+09]. Nagappan et al. [NMB08] found that organizational metrics are even more predictive of software-quality factors than traditional source-code metrics. Shin et al. [SMW+11] showed that developer-activity metrics extracted from networks are predictive of security vulnerabilities. When it comes to developers roles, Joblin et al. [JAH+17] found that the structural position of developers within the network is reflective of developer perception. These studies show evidence that developer networks contain rich structure that is related to several highly relevant socio-technical dimensions of a software project. Inspired by these results, our primary focus is on achieving a deeper understanding what fundamental structures exist, how they evolve over time, and how developers embed within them.

Determining whether the edge formation in developer networks is driven primarily by a purely random process or by non-random organizational principles has received some attention. Bird et al. [BPD+08] found that developer communication networks contain a latent community structure that is consistent with non-random edge formation. Similarly, in Chapter 4, we identified latent group structures in communication and co-editing. In developer networks constructed from commit activity, Joblin et al. [JMA+15] found that developers form communities that are statistically significant and align with developers' perceptions of team collaboration. Also, the scale-freeness property appears to be pervasive among projects that achieve sustained long-term growth, and there is initial evidence of hierarchy existing in OSS projects [JAM17]. Given that hierarchy functions as a unifying principle of two other important structures, communities and scale-freeness, it is imperative to better understand the role hierarchy plays is OSS projects.

5.2 Research Questions

Before we describe our method, we discuss the research questions that motivated us to develop the method in the first place.

> **RQ$_1$:** Are there patterns in terms of structure and evolution that exist in OSS projects with respect to hierarchy?

Hierarchy and role stratification emerge in times of project growth and increased coordination overhead [HM06; TLV13; YK03], and organizational structure is closely related to project sustainability and scalability [ZCM+17]. As a consequence, the fundamental organizing principles at play must evolve throughout the project life cycle. It remains an open question, however, whether hierarchical, non-hierarchical, and hybrid structures are a pervasive phenomenon governing organization in OSS projects and at which points in the project life cycle these structures exist. Answering RQ_1 sheds light on hierarchy and hybrid structure as a global network property that co-evolves with the different project phases and growth regimes. Since each type of organizational structure exhibits certain strengths and weaknesses, by knowing what organizational structures exist and when, we would be able to asses whether the observed structures make appropriate accommodations to the current project conditions.

> **RQ_2:** How does an individual developer's position in the hierarchy change over time?

While the perspective of RQ_1 is rather global, we now peer through the lens of individual developers to understand their personal journey through the organizational structure. Similar to the organizational structure of a project, the roles of developers evolve along with their activity in the project [ZM10]. Understanding the social dynamics in OSS projects requires understanding the relationship between on-boarding of developers and their later roles [CDO+12; ZM12] and the determinants of the chances to reach an influential position [CLL+17; ZM11]. In our study, we look for typical "trajectories" of developers through a project's hierarchy. Clearly, different roles imply different experience levels, consistency of involvement, and level of commitment. Since certain organizational structures are not well suited for high volatility (e.g., hierarchy), the inconsistent involvement exhibited by some roles posses a risk, unless the roles are organized differently. By answering this question, we are able to observe how large-scale and successful OSS projects integrate new developers and how their progression is reflected in their structural position. This information is useful to identify problematic on-boarding, off-boarding, and developmental progression practices occurring in a project, and to apply corrective measures to ensure a higher probability of success.

> **RQ_3:** How are a developer's contacts distributed over the organizational structure of an OSS project?

One way how network hierarchy can benefit project quality and success is through improved communication and information flow. Consequently, it is important to understand who a developer's social contacts and cooperation partners are. Canfora et al. [CDO+12] investigated who is responsible for on-boarding new community members in OSS projects. They found that whether a developer's first contact is an isolated or a well-integrated community member has an important influence on information flow and knowledge sharing. Afsar and Badir [AB15] and Zhou and Mockus [ZM15] provide more evidence for knowledge sharing of colleagues and especially newcomers during their on-boarding phase. Steinmacher et al. [SSG14] analyzed the extent to which social contacts into the group of an OSS community's core developers simplify on-boarding into new projects. We expect that developers who communicate early with other developers from the top of the hierarchy will eventually rise in the hierarchy, which we will verify by tracking their positions in the hierarchy over time.

> **RQ$_4$:** How do tenure and programming activity of developers affect their position in the project's organizational structure?

Since different positions in the organizational structure enable different functions, in a healthy project, the position a developer has should ideally support the developer's role and should not expose the project to unnecessary risk. In OSS projects, important developers are often responsible for both, coordination and the bulk of the programming work [EZZ19; RGH09; YK03]. By analyzing e-mail data and data from issue trackers in combination with commit data, we are able to investigate the relationship of coordination and programming tasks with respect to a developer's position in the hierarchical structure of the project. The entire process usually takes time: Developers become core developers and project leaders only with sufficient experience in the project [BGD+07; ZM12]. In combination, these aspects may explain why tenure has only a weak relation to the number of code contributions: The number of code contributions increases only in the starting phase of a project and caps after around 3 years [ZM10]. It is beneficial to understand the relationship between function and structural position in large-scale OSS projects, because it helps us to establish practices that are conducive to successful outcomes. For example, if developers with primarily coordination tasks are positioned outside of the hierarchy, that could be an indication that there is a mismatch between the function and structural position. In that case, policies and practices should be revised to induce a healthier project structure.

5.3 Methodology

We now describe the data and developer networks that we use, as well as the corresponding statistical methods that we use to identify the hierarchical and the non-hierarchical part.

5.3.1 Data Extraction & Construction of Developer Networks

For this study, we extract communication data from OSS projects and build developer networks from it. To cover two different kinds of communication channels, we investigate projects that use a mailing list for communication among developers, as well as projects that use the GitHub issue tracker for developer communication. We extract mailing-list data from Gmane and issue data (including pull-request data) from GitHub's official REST API[1], using the tools nntp2mbox, Codeface, and GitHubWrapper, which we have introduced in Section 2.4. As previous research has shown that GitHub issue data could be distorted by bots that automatically comment on issues but are not human beings [GLD+20; WSS+18], we detect bots with the tool BoDeGHa and remove comments created by bots from the issue discussions, as described in Section 2.4.

Depending on whether a project uses mailing lists or issue discussions, we either build *mail* networks or *issue* networks (see Section 2.3.1). For the construction of issue networks, in this study, we only consider comments in the issue and pull-request discussions. That is, we consider the initial comment authored by the issue creator, all subsequent comments to this

1 `https://docs.github.com/en/rest/` (accessed at 2022-03-14)

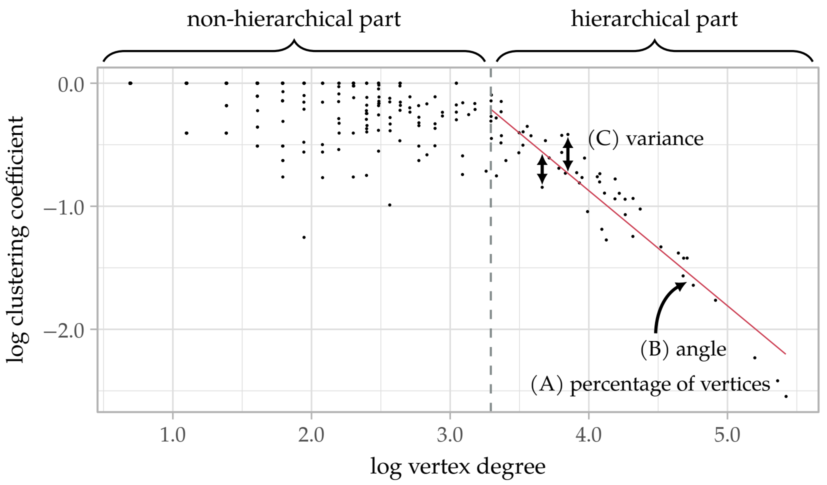


Figure 5.3: The plot shows the log-transformed vertex degree and the log-transformed clustering coefficient of all developers who were active in this time range. Each dot represents one developer. There is a division into a hierarchical and a non-hierarchical part (dashed line). We mark variance (C), angle (B), and percentage of vertices in the hierarchical part (A).

issue, as well as pull-request review comments and all their replies.[2] Independently of the network type, we build undirected, simplified networks, since the hierarchy measure we use is defined on undirected networks without edge weights [RB03]. To build and analyze the developer networks, we use the library CORONET, which we have introduced in Section 2.4.

To capture the temporal dimension of a project's history and to allow for smooth transitions between adjacent time windows, we use *sliding* windows with a length of 6 months (i.e., an overlap of 3 months), as explained in Section 2.3.2. This choice balances between fluctuation and noise in short time ranges (prohibiting us from observing any non-spurious patterns, plus randomly losing communication activity that happens across time-range borders) and losing detail in long ranges due to aggregation [MW11].[3]

To investigate the relationship between organizational structure and programming activity in RQ_4, we also extract the commit metadata of the projects from the corresponding GIT repositories using CODEFACE. We match the commit metadata with the communication data of the corresponding time ranges using CORONET.

5.3.2 Typical Structure and Evolution

To answer RQ_1, we visualize, for each network, the hierarchical position of all developers in one plot, as we do in Figure 5.3. The plot covers a single time range and shows for every developer i the vertex degree k_i (i.e., the number of developers a developer is connected with) on the x axis (using logarithmic transformation) and the corresponding local clustering

2 Other events in GitHub issues, such as labeling or merging, are ignored in this study, to focus on actual communication activities only. This is in line with the construction method of mail networks, which are by construction based on communication activities only.

3 We also performed a sensitivity analysis and tried sliding windows with a length of 3, 6, 9, and 12 months. However, in this thesis, we present and discuss only the results of using 6-months time windows, as justified above. Nonetheless, we provide all results for using the remaining time windows on our supplementary website.

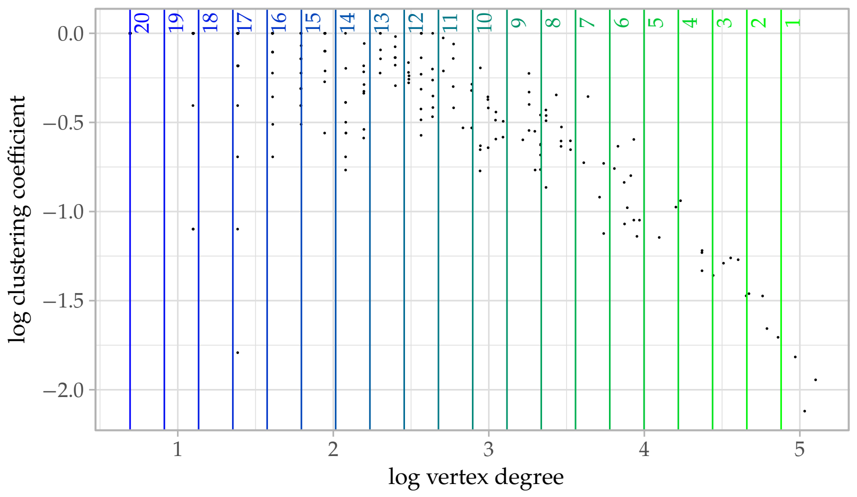


Figure 5.4: Candidate breakpoints for a single time range of project LLVM (2008-06 to 2008-12). Each vertical line denotes one possible split into a hierarchical part and a non-hierarchical part.

coefficient c_i (i.e., the connectedness of the neighbors of a vertex) in this time range on the y axis (using logarithmic transformation). To the right of the dashed line in Figure 5.3, we show the relationship between $\log(c_i)$ and $\log(k_i)$ that is typical for a hierarchical network: a linear relationship with negative slope [RB03]. To the left of the dashed line, the relationship appears fundamentally different in that there is certainly no linear relationship and even appears to be no relationship between $\log(c_i)$ and $\log(k_i)$. The absence of a linear relationship with a negative slope indicates that hierarchical organization is not present and is consistent with the random network introduced in Figure 5.2. Determining the hierarchical and non-hierarchical parts corresponds to identifying a breakpoint between the linearly related segment and the remaining unrelated segment (i.e., identifying an optimal position for the dotted line in Figure 5.3). In Figure 5.4, we illustrate the decision space for this problem, where each vertical line corresponds to a candidate breakpoint. Our method for automating the identification of this breakpoint relies on a mixed approach of human labeling and combinatorial optimization. The human labeling is used to learn rules for trimming the candidate space, making the combinatorial optimization procedure more efficient. As a secondary use of the human labeled data, we test how well our automated method generalizes to decisions made by human annotators that were not used to find parameters for the method.

Manual Data Labeling We used human annotation to obtain labels for 289 randomly selected developer networks from a subset of our subject projects (see Section 5.4.1). In this case, a labeled data point corresponds to a network and the position of a breakpoint for decomposing the network into hierarchical and non-hierarchical parts. The division into a hierarchical and non-hierarchical part was done in a majority vote based on visual inspection by three of the researchers that were involved in this study; no specialized domain knowledge was required for this task. As no bias should be introduced into this judgment, no information on the project and its members was used during the process.

To perform this labeling task, we showed a figure with 20 candidate breakpoints (see Figure 5.4) to the human annotators and asked them to select one. We formed candidate groups by dividing the range of log vertex degrees into 20 buckets (intervals) of equal length $b_1..b_{20}$. This choice provided a reasonable compromise between granularity and computational cost. The first *candidate group* for the hierarchical part contained the developers in the bucket with the highest vertex degrees (b_1), with a minimum of two developers, while the remaining 19 buckets ($\bigcup_{i=2..20} b_i$) formed the non-hierarchical part. The second candidate group contained developers of the two buckets with the highest vertex degrees ($b_1 \cup b_2$), while the developers of the remaining 18 buckets ($\bigcup_{i=3..20} b_i$) were seen as the non-hierarchical part. This was continued until all developers are included in the candidate group for the hierarchical part, resulting in 20 candidate groups in total. The annotation instructions indicated that the breakpoint should be chosen such that points to the right exhibit a linear relationship with a negative slope and points to the left do not. Each network to be labeled was shown to multiple annotators, and an average of the chosen buckets was taken afterwards. For the example in Figure 5.4, the annotators chose the buckets b_{10}, b_{11}, and b_{13}, resulting in b_{11} as the average bucket in this case.

We used 80% (231) of the labeled examples for fitting parameters in our automated method, described in detail below. The remaining 20% (58) of the labeled examples were used to test the agreement between our automated method and human annotators on networks not used during parameter fitting. The results indicate that the automated method and human annotation differ on average by 10% (i.e., an error of two buckets). For the example in Figure 5.4, our automated method (which we describe in what follows) selects b_{13}.

Combinatorial Optimization In what follows, we describe our automated method for determining the hierarchical and the non-hierarchical part of a developer network.

Step 1: Compute metrics of the candidate groups. To choose the optimal split of the developers into a hierarchical and a non-hierarchical part, we start from 20 equally-sized candidate groups, which was a reasonable compromise between granularity and computational cost for our dataset. Three metrics (also visualized in Figure 5.3 as dashed line and arrows) for every candidate group served as input to an optimization procedure:

(A) the percentage of vertices of the hierarchical part (*perc*);
(B) the angle between regression line (red solid line in Figure 5.3) and x axis (*angle*);
(C) and the residual variance of the regression fit (*var*).

The percentage of developers in the hierarchical part (*perc*) is defined as the number of developers (black dots) to the right of the dashed line divided by the number of all developers in the plot (all developers in the time range with two or more interactions). The solid line is the fitted regression line from a least-squares single linear regression of the log clustering coefficient on the log vertex degree. The *angle* between this line and the x axis (the bent arrow) is related to the hierarchical part, such that a higher angle indicates a stronger hierarchical structure [JAM17; RB03]. If this angle is zero, no hierarchy is present. The goal is to find the largest possible angle. We compute the residual variance (*var*) for the hierarchical part of the candidate group as the average squared deviation of the black dots right of the dashed line (illustrated by the double-sided arrows in Figure 5.3). The regression line should be as close to each black dot (developer) as possible. Thus, the variance should be as low as possible. As result of Step 1, we know *perc*, *angle*, and *var* for each of the 20 candidate groups.

Step 2: Apply thresholds to candidate groups. We aim at identifying the most suitable set of developers comprising the hierarchical part. To this end, the decision on whether to keep a candidate group or not is based on thresholds that we determined from the manual data labeling (as described above in Section 5.3.2), which we have performed on a sample of our subject projects, to filter degenerate cases (e.g., a large variance that blurs the hierarchical structure). Hence, based on all the selected buckets that resulted from the manual data labeling, we computed the following thresholds, which we now apply to all candidate groups:

(A) the hierarchical part must contain more than 5% of the developers ($perc > 5\%$);
(B) the angle between x axis and regression line must be greater than 35° ($angle > 35°$);
(C) and the residual variance of the regression fit must be smaller than 0.5 ($var < 0.5$).

All candidate groups that satisfy these three requirements are the input for the third step. Formally, we denote the set of remaining candidate groups for time range $t \in 1..T$ as S_t.

Step 3: Select optimal candidate group. We select one of the candidate groups $s \in S_t$ per time range that best describes the hierarchical and the (remaining) non-hierarchical part. To this end, we compute $\overline{S} = \bigcup_{t \in 1..T} S_t$, the union of the sets of candidate groups of all time ranges per project. As the range of the three criteria varies from project to project, we first standardized the values from *Step 2* project-wise by subtracting the project-specific means of the criteria (Equation 5.1) and by dividing by the standard deviation of the criteria (Equation 5.2). If m_s represents any of the three measures for candidate group $s \in \overline{S}$ in time range $t \in 1, \ldots, T$, and $N = |\overline{S}|$ is the number of all candidate groups for one project across all time ranges, the calculus is:

$$\overline{m} = \frac{1}{N} \cdot \Big(\sum_{s \in \overline{S}} m_s \Big) \tag{5.1}$$

$$sd = \sqrt{\frac{1}{N-1} \cdot \Big(\sum_{s \in \overline{S}} (m_s - \overline{m})^2 \Big)} \tag{5.2}$$

$$m_s^{std} = \frac{m_s - \overline{m}}{sd}. \tag{5.3}$$

This results in a standardized percentage measure $perc_s^{std}$, angle $angle_s^{std}$, and residual variance var_s^{std}. To solve the optimization problem of finding the best candidate group, we use the three standardized measures. We aggregate them by computing a weighted sum, giving higher priority (three times) to the percentage of developers in the hierarchical part. We determined this priority based on the manual data labeling, which we have performed on a sample of our subject projects. Then, one candidate $s \in S_t$ is selected per time range $t \in 1..T$, such that the weighted sum of the three criteria is maximal:

$$\underset{s \in S_t}{\arg\max}\ (3 \cdot perc_s^{std} + angle_s^{std} - var_s^{std}) \tag{5.4}$$

This way, our method favors high values for criteria (A) and (B), but penalizes high values for criterion (C). This procedure results in one split per time range and project.

Synopsis. In Algorithm 5.1, we summarize the above described steps of our automated method for determining the hierarchical and the non-hierarchical part of a developer network.

Algorithm 5.1 Division of developers into a hierarchical and a non-hierarchical part

```
Input: networks $net_t$ for each time range $t \in 1..T$
 1: for each $t \in 1..T$ do
 2:
 3:     ▷ compute basic statistics
 4:     for each developer $i$ in $net_t$ do
 5:         $k_i \leftarrow$ vertex degree of $i$
 6:         $n_i \leftarrow$ number of edges between the neighbors of $i$
 7:
 8:         ▷ clustering coefficient
 9:         $c_i \leftarrow 2 \cdot n_i / (k_i \cdot (k_i - 1))$
10:     end for
11:
12:     ▷ compute candidate groups $S_t$ for hierarchy
13:     divide range of $\log(k_i)$ into 20 buckets $b_1..b_{20}$
14:     $s \leftarrow b_1$, $S_t \leftarrow \{s\}$
15:     for each $z \in 2..20$ do
16:         $s \leftarrow s \cup b_z$, $S_t \leftarrow S_t \cup \{s\}$
17:     end for
18:
19:     ▷ Step 1
20:     for each $s \in S_t$ do
21:         compute $perc_s$, $angle_s$, and $var_s$
22:
23:         ▷ Step 2
24:         if $\neg(perc_s > 0.05 \land angle_s > 35° \land var_s < 0.5)$ then
25:             $S_t \leftarrow S_t \setminus \{s\}$
26:         end if
27:     end for
28:
29: end for
30:
31: ▷ Step 3
32: $\overline{S} \leftarrow \bigcup_{t \in 1..T} S_t$
33: for each $s \in \overline{S}$ do
34:     compute standardized values $var_s^{std}$, $perc_s^{std}$, $angle_s^{std}$
35: end for
36:
37: ▷ selection of optimal candidate group $s_t$
38: for each $t \in 1..T$ do
39:     $s_t \leftarrow \arg\max_{s \in S_t} (3 \cdot perc_s^{std} + angle_s^{std} - var_s^{std})$
40: end for
Output: sets of developers of the hierarchical part $s_t$ for each time range $t \in 1..T$
```

5.3.3 Change of Position in Hierarchy

To address RQ_2, we track how a developer's position in the organizational structure changes over time. To track the position of individual developers in the hierarchy, we compare their position across different time ranges. For this purpose, to prevent distorting effects, we normalize the log vertex degree and the log clustering coefficient to analyze the position

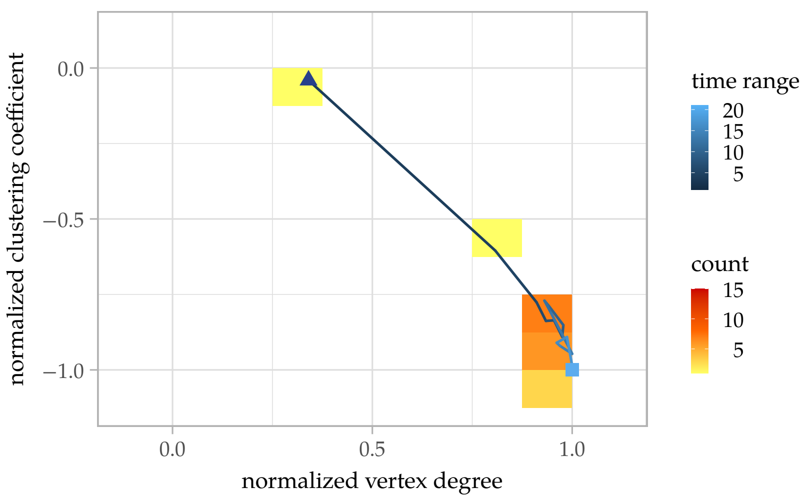


Figure 5.5: The considered developer of the project Node.js (developer 874) starts in the non-hierarchical part (dark blue triangle) with a low number of contacts and a high clustering coefficient. Later, the developer moves to the hierarchical part, having many contacts and a low clustering coefficient, until the developer leaves the project (light blue rectangle). The color of the big rectangles in the background denotes the count of how many time ranges the developer has held a certain position; the darker the rectangle, the longer the developer has held the position.

independently of the network size (i.e., number of developers) across the networks of the entire project history. The clustering coefficient ranges between $[0, 1]$, so its log ranges from $-\infty$ to 0. We normalize the smallest clustering coefficient over a time range and project to a value of -1, by dividing all log clustering coefficients by the absolute value of the smallest log clustering coefficient. This way, the developer with the highest vertex degree and the lowest clustering coefficient of a time range is always on the same position in the hierarchy plot, irrespective of the network size. The closer the *normalized clustering coefficient* is to -1, the more important is the respective developer's role as hub. The log vertex degree is a positive number. We normalize it to a value in $[0, 1]$ by dividing all log vertex degrees by the highest log vertex degree in the given time range. Thus, the developer with the highest vertex degree has a *normalized vertex degree* of 1 (see Figure 5.5).

We describe the developer's *change of position* as changes in the developer's normalized position in the organizational structure. In Figure 5.5, we show such a change with a trajectory and a heat map for a developer of the project Node.js. The considered developer starts (dark blue triangle) with a low vertex degree and a high clustering coefficient. Then, the developer's vertex degree increases while the clustering coefficient decreases (movement to the hierarchical part), until the developer reaches a very high position (bottom right of the plot) in the project community, which is stable for several time ranges. Thus, the developer has acquired many neighbors who do not interact a lot with each other, until the developer leaves the project (light blue rectangle).

We restrict our analysis to the 10 most active developers per project, as understanding the typical contribution and activity cycle of them allows us to analyze the on- and off-boarding process of socially active developers. To detect the 10 most active developers, we compute

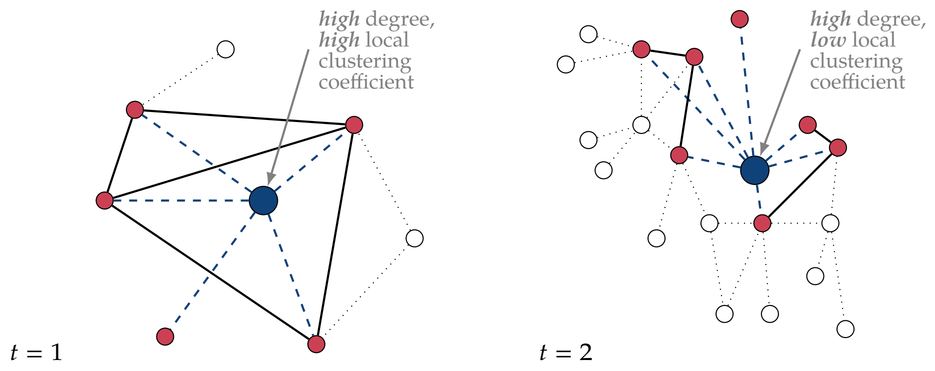

high degree,
high local clustering coefficient
high degree,
low local clustering coefficient
$t = 1$
$t = 2$

Figure 5.6: Neighborhood of a selected developer (blue, large vertex) at two different time ranges ($t = 1$ and $t = 2$). The red vertices (which are connected to the selected developer via dashed lines) form the neighborhood of the selected developer. The proportion of edges among the neighbors (solid lines) determines the local clustering coefficient of the selected developer. As the local clustering coefficient changes over time, also the position of the selected developer in the hierarchy changes over time.

the total number of interaction partners per developer over the project history. To achieve a more complete perspective, we also select 10 developers randomly and analyze their changes of position. In addition, we also provide descriptive statistics for the selected developers (number of commits, number of e-mails or issue comments, respectively, the number of active time ranges, and how many of these time ranges the developer was part of the hierarchical part), to give an impression of their activity.

5.3.4 Developers' Neighborhood

To answer RQ_3, we extract the direct neighbors (i.e., first-order neighborhood) of the developers selected in RQ_2 and analyze the distribution of the neighbors regarding their position in the hierarchical or non-hierarchical part. Since the networks evolve over time, we extract the neighborhood of a selected developer separately for each time range (see Figure 5.6) and investigate how the distribution of neighbors regarding their position in the hierarchical or non-hierarchical part changes over time. Based on the network type, a neighbor is a developer who participated in, at least, one common communication activity (i.e., an e-mail thread for mail networks or an issue discussion for issue networks, respectively) within the 6-months time range on which the network has been constructed.

5.3.5 Tenure and Programming Activity

To answer RQ_4, for each developer and time range, we compute tenure as the time between the first communication activity of a developer and the end date of the investigated time range. In addition, to investigate programming activity, we compute the number of files edited by each developer within a time range. To that aim, we extract the number of edited files per developer from the version-control system of the project. For both, tenure and programming activity, we compare the developers of the hierarchical part and the non-hierarchical part, globally as well as their trends over time. We use a one-tailed, unpaired Mann-Whitney U test for the global comparisons and report the corresponding p-values as well as Cliff's Delta d, which quantifies the effect size corresponding to the statistical test.

5.4 Longitudinal Study

To answer our research questions, we conducted an exploratory, longitudinal study on 20 popular and widely-used OSS projects. In this section, we summarize and discuss the most important results and observations of our study, and we derive a number of hypotheses to guide further work. As we generated a large amount of data, we present only selected cases and representative figures here (e.g., we use only results from the 6-months time ranges, as justified in Section 5.3.1). The remaining data are available on our supplementary website.[4]

5.4.1 Subject Projects

For our study, we selected 20 subject projects. Since OSS projects are of different age and use a wide array of communication channels [EGW22; KGB+18; SSF+17; TNK+19], we focus on projects that either use a mailing list or an issue tracker as their main communication channel, from which we construct developer networks. Half of our subject projects uses a mailing list as their main communication channel (see the upper half in Table 5.1), and the other half of them mainly uses GitHub issues (see the lower half in Table 5.1); none of our subject projects uses both communication channels simultaneously. Both kinds of communication channels that we investigate, developer mailing lists and GitHub issues, are used for public, technical discussions among developers and for reviewing code changes (in terms of patches on mailing lists, in terms of pull requests on GitHub issues). Hence, both communication channels contain similar content and are used for similar purposes, which is why only either of the channels is used in each of our subject projects.

We cover projects from various domains (including compilers and virtualization software), programming languages (e.g., C, Python, JavaScript), and sizes (from 132 kLOC to 10 988 kLOC; see Table 5.1). To reduce bias due to noisy and incorrect data, we limited our selection of projects: For projects with publicly accessible mailing lists, we only selected projects that have already been studied in Chapter 4 or in the related literature [JAH+17; JAM17; JMA+15]. For the selection of projects that use the GitHub issue tracker, we considered the list of most starred GitHub projects in 2020.[5]

In Table 5.1, we show the maximum and minimum number of developers within a time range per project, as well as the number of developers at a project's start and end times. In all projects, the number of developers increases, reaches a peak, and then decreases. An exception is project Wine. There, we observe only a decrease. The typical behavior happens in four patterns: (1) a slow and steady increase is followed by a short decrease: for Electron, LLVM, QEMU, and React; (2) a steeper increase is followed by a slower and smooth decrease (with possible bumps): for Angular, Atom, Django, GCC, Moby, Node.js, ownCloud, Qt, and

4 `https://hierarchypaper.bitbucket.io/public/`
5 `https://www.attosol.com/top-50-projects-on-github-2020/` (accessed at 2020-09-09)

Table 5.1: Overview of our subject projects.

Project	Period[1]	# Global[2]	# Max[2]	# Min[2]	# First[2]	# Last[2]	kLOC[3]	Project Domain
Django	2005–2017	4 458	601	139	174	139	657	Web application framework
FFmpeg	2002–2017	5 668	531	213	213	285	1 431	Video/audio conversion
GCC	2000–2018	9 086	1 295	2	2	262	10 988	C compiler suite
Git	2004–2017	9 151	951	2	2	579	742	Version control
LLVM	2002–2017	6 230	982	2	27	2	2 883	Compiler framework
ownCloud	2009–2018	1 487	389	26	26	30	588	File hosting
QEMU	2003–2016	7 131	781	38	38	551	1 430	Visualization software
Qt	2008–2018	1 312	358	3	11	145	7 461	GUI toolkit
U-Boot	2000–2017	7 684	735	9	9	449	1 942	Boot loader
Wine	2002–2017	3 436	424	12	298	12	4 864	Compatibility layer
Angular	2014–2020	22 304	4 248	5	5	2 177	1 050	Web development platform
Atom	2012–2020	20 646	3 655	8	8	626	242	Text editor
Bootstrap	2011–2021	23 602	3 413	631	1 782	631	132	Web frontend framework
Electron	2013–2020	15 017	2 139	15	15	1 622	225	Application framework
Flutter	2015–2020	33 800	11 493	27	27	10 325	1 145	UI development kit
Moby	2013–2020	27 777	4 691	386	386	1 151	1 636	Software containerization
Node.js	2014–2020	12 165	2 197	865	1 422	865	7 234	JavaScript runtime
React	2013–2020	15 220	2 062	135	135	958	402	JavaScript library
TypeScript	2014–2020	17 703	3 165	491	491	2 573	3 350	JavaScript language
webpack	2012–2020	12 324	2 199	9	9	814	200	Bundler for modules

[1] Period: time period of availability of e-mail data (upper 10 projects) and issue data (lower 10 projects)
[2] # Global: number of active developers; # Max, # Min: maximal and minimal number of active developers (incl. developers with 1 or 0 contacts) in a 6-months time range; # First, # Last: number of active developers in the first and last investigated 6-months time range
[3] kLOC: number of lines of code (LOC) in thousands, including comments and blank lines, at the end of the investigated time period

webpack; (3) increase, decrease, increase: for Bootstrap, FFmpeg, and Git; (4) increase: for Flutter, TypeScript, and U-Boot. An example of the most frequent pattern (2) is project GCC (see Figure 5.7 (top)).

5.4.2 Typical Structure and Evolution

To address RQ_1, we investigate whether there are patterns in terms of structure and evolution that exist in OSS projects with respect to hierarchy.

Results With an increasing number of developers, the proportion of developers in the hierarchical part decreases in most projects; and, with a decreasing number of developers, the proportion of developers in the hierarchical part increases (Angular, Atom, Bootstrap, Electron, Flutter, GCC, Git, LLVM, Moby, Node.js, ownCloud, Qt, QEMU, React, TypeScript, U-Boot, and webpack). We illustrate this in Figure 5.7 (bottom) for GCC: The percentage of

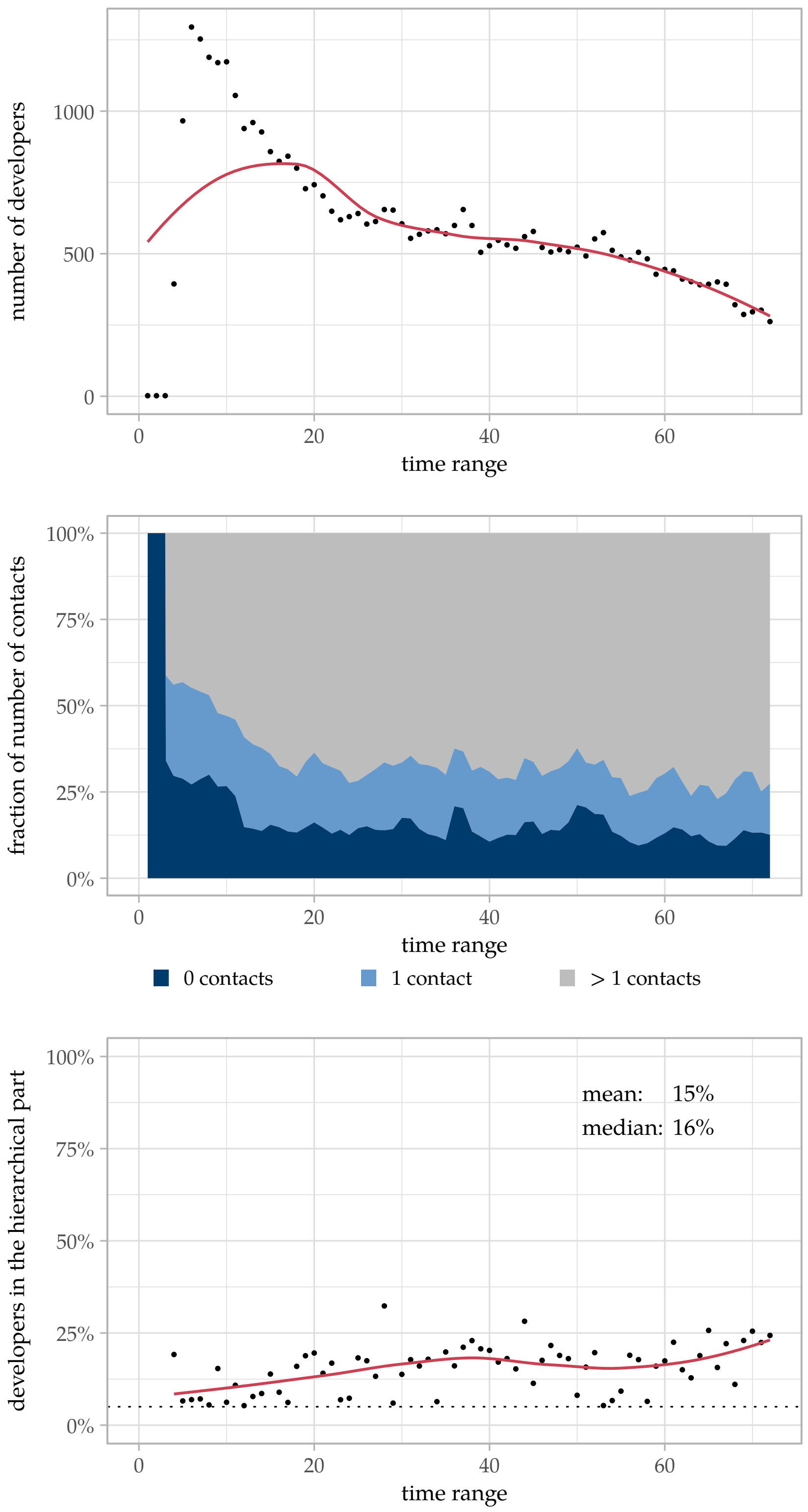


Figure 5.7: Project GCC shows a typical evolution pattern: The number of developers increases, and after a certain period of time, the number decreases (top). The fractions of developers with no, one, or more than one contacts become stable (middle). The fraction of developers in the hierarchical part slowly increases (bottom).

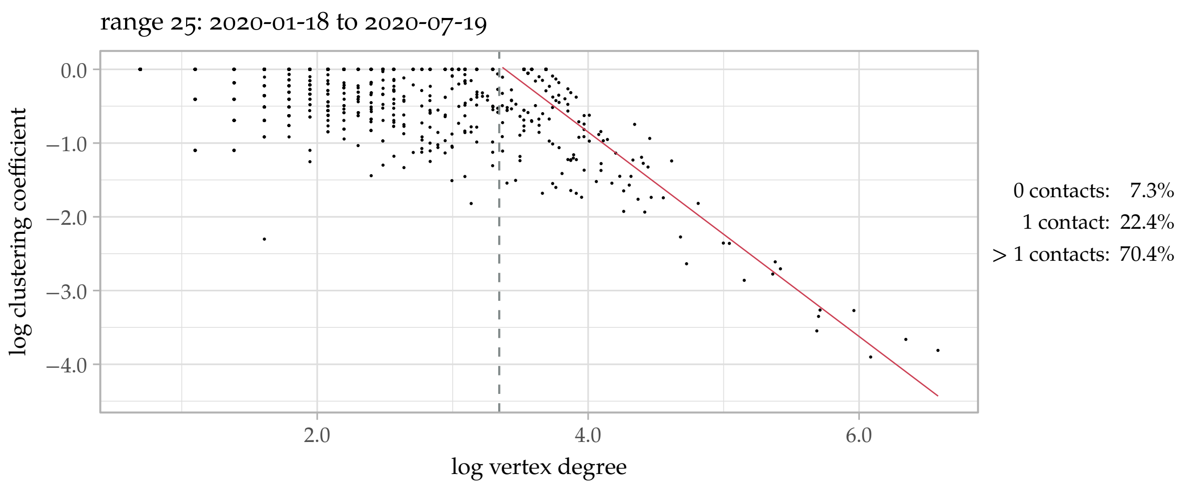


Figure 5.8: Hierarchical structure of the 25th analyzed 6-months time range of project ANGULAR: The plot shows the log-transformed vertex degree and the log-transformed clustering coefficient of all developers who participated in the issue discussions of this time range. We identified a hierarchical structure (to the right of the dashed line).

developers in the hierarchical part grows from 7% to 25%, while the number of developers falls from around 1 300 developers to around 260 developers. For FFMPEG and DJANGO, the portion of the hierarchy and the number of developers are independent of each other. The portion of hierarchy of WINE is mainly stable, but in the end, we find an increase of the portion.

Only in a low number of the analyzed time ranges, we do not find a hierarchical structure: ANGULAR (range 1), ATOM (range 1), FLUTTER (ranges 16 and 17), GCC (ranges 1, 2, and 3), GIT (ranges 1 and 2), OWNCLOUD (range 31), QT (ranges 1–6), U-BOOT (range 1), WEBPACK (range 1), and WINE (range 61). This mostly happens in the very first time ranges of a project, where only very few developers are communicating yet, ending up in a loosely connected network where most developers have less than three connections to others.

Nonetheless, for almost all ranges of the above stated projects and all ranges for the projects BOOTSTRAP, DJANGO, ELECTRON, FFMPEG, LLVM, MOBY, NODE.JS, QEMU, REACT, and TYPESCRIPT, we are able to identify a hierarchical structure for all analyzed time ranges. To provide an example, we show the hierarchical part of project ANGULAR in Figure 5.8 (to the right of the dashed line). Over time, the residual variance of the regression fit and the angle between regression line and x axis (see Section 5.3.2) stay relatively stable for each project.

Answer to RQ$_1$: We observe the presence of a hierarchical part for all analyzed projects. A pervasive phenomenon is that the proportion of developers in the hierarchy changes over time: At the beginning, when projects have few developers, almost all developers tend to be positioned within the hierarchical part; as the project matures and grows, the vast majority of developers exist outside of a hierarchy (up to 90%). Thus, we see evidence of temporal patterns that indicate a fundamental shift in the organizing principles at play in OSS projects. The fundamental shift is from hierarchy as a global organizing principle at an early project stage to a local hierarchical part with the vast majority not hierarchically organized in later project stages.

Discussion As we were able to identify a hierarchical structure in almost all time ranges for all subject projects, independent of the number of developers and independent of the communication channel (issue tracker or mailing list), our method on decomposing developer networks into a hierarchical and a non-hierarchical part is generalizable to projects of different sizes, different ages, different domains, and different communication channels. Our findings support previous indications that successful OSS projects develop a hybrid organizational structure composed of a hierarchical and a non-hierarchical part, with most of the developers being part of the non-hierarchical part.

The presence of a hierarchical part tends to be unaffected by variations in the numbers of developers of a project; variation tends to be limited to fluctuations in its relative size as the project matures. Developers who enter or leave the hierarchical part change its composition. Despite this developer turnover, the slow change in residual variance and in the angle between the regression line (red solid line in Figure 5.8) and the x axis over time, as indicators for stable hierarchical structure, suggests that the subject projects have a stable organizational structure. This finding is in line with the hypothesis that the hierarchical part is principally responsible for coordination supporting information exchange [JAH+17]. In this case, one would indeed expect that successful projects achieve a stable hierarchy, since large structural shifts disrupt coordination.

5.4.3 Change of Position in Hierarchy

To address RQ_2, we analyze for individual developers how their position in the hierarchy changes over time.

Results In Figure 5.9, we show how the position of an exemplary developer changes during the evolution of project LLVM. The developer starts in the upper left of the plot, that is, at the bottom of the hierarchy or already in the non-hierarchical part. Then, the clustering coefficient decreases as the developer moves down in the plot toward the hierarchical part, where the developer stays for around 12 time ranges. Then, the developer moves back to the non-hierarchical part and potentially leaves the project (if this is not yet the end of our analyzed time period).

We find notable patterns of positional change, which we summarize in Table 5.2 for mail networks and in Table 5.3 for issue networks. In total, we analyze 200 developers per data source (i.e., 200 developers for issue networks and 200 developers for mail networks)—the 10 developers with maximum vertex degree and 10 random developers per project, as described in Section 5.3.3. We provide descriptive statistics for both, the most active and the randomly selected developers, in Table 5.4. In general, the majority of the randomly selected developers contributed no commit to the project and only few e-mails or issue comments, whereas the most active developers were not only highly active in communicating, but were also highly active code contributors. (Project GCC is an exception, where even none of the most active communicators contributed any commit to the source code; we discuss this phenomenon further below in Section 5.4.5.) As expected, the most active developers appear to be active in more time ranges than the randomly selected developers do. In line with that, the most active developers are mostly part of the hierarchical part, whereas the randomly selected developers are only rarely part of the hierarchical part.

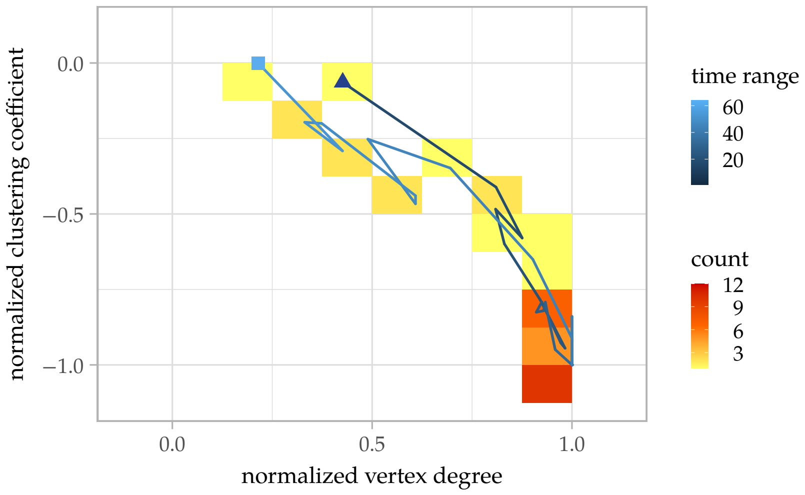


Figure 5.9: The considered developer of the project LLVM (developer 1610) starts with a low number of contacts and a high clustering coefficient in the non-hierarchical part (dark blue triangle). The developer has many contacts in the middle of the project's analyzed time ranges, then they move to back to the non-hierarchical part, until they leave the project (light blue rectangle). Hence, the corresponding movement pattern for this developer is "down → up". The color of the big rectangles in the background denotes the count of how many time ranges the developer has held a certain position; the darker the rectangle, the longer the developer has held the position.

The mail and issue networks exhibit largely similar movement patterns for the selected developers. Thus, we focus on the details of mail networks. The movement patterns in Table 5.2 describe different starting points and directions of position changes in the organizational structure. The most frequent pattern occurs in all projects: 40 out of 100 most active developers start at the non-hierarchical part's upper left region in the hierarchy plot, then move down to the upper levels of the hierarchical part (lower right), to finally return to the non-hierarchical part again (pattern "down → up"). An example of this pattern is developer 1610 of the project LLVM (see Figure 5.9). The two second-most frequent patterns (18 out of 100) describe developers who move from the non-hierarchical part to the upper levels of the hierarchical part (that is, down to the right in the hierarchy plot, pattern "down") and developers moving in the opposite direction (18 out of 100), that is, they start in the hierarchical part and then move to the non-hierarchical part (pattern "up"). The five remaining patterns play only a secondary role and do not occur often. In the end, we find 11 developers over all projects who have other (individual) movement patterns.

For the randomly selected developers, we find the same patterns, but with different frequencies. Often, developers remain relatively constant in one area in the hierarchy (35 out of 99, pattern "constant") and they move only slightly. We find also that developers move horizontally (i.e., they have more neighbors, but the connectivity between the neighbors stays constant, 18 out of 99, "horizontal") and remain active only for few time ranges.

Table 5.2: Frequency of directions of positional change in the hierarchy for the 10 most active/10 randomly selected developers for mail networks.

Movement Pattern	Django	FFmpeg	GCC	Git	LLVM	ownCloud[1]	QEMU	Qt	U-Boot	Wine	$\sum$	$\sum_{\text{all devs}}$
Down → up	5/1	4/–	2/–	4/1	6/2	7/1	6/2	–/4	3/3	3/3	40/17	57
Constant	–/5	1/3	–/5	1/2	–/4	–/–	–/6	2/3	–/5	1/2	5/35	40
Down	2/1	1/–	1/–	2/–	2/1	1/3	3/–	1/–	5/–	–/1	18/6	24
Up	1/–	3/–	1/–	1/–	–/–	1/2	1/–	6/1	1/–	3/2	18/5	23
Horizontal	–/–	–/5	–/1	–/5	–/1	–/1	–/2	–/1	–/2	–/–	–/18	18
Down → up → down	2/–	–/–	1/–	2/1	1/–	1/–	–/–	–/–	–/–	–/–	7/1	8
Up → down	–/1	–/–	1/1	–/1	–/–	–/–	–/–	–/–	–/–	–/–	1/3	4
Up → down → up	–/1	–/–	–/–	–/–	–/–	–/1	–/–	–/–	–/–	–/–	2/–	2
Other	–/1	1/2	4/3	–/–	1/2	–/1	–/–	1/1	1/–	3/2	11/12	23

[1] There are only 9 randomly selected developers in ownCloud.

Table 5.3: Frequency of directions of positional change in the hierarchy for the 10 most active/10 randomly selected developers for issue networks.

Movement Pattern	Angular	Atom	Bootstrap	Electron	Flutter	Moby	Node.js	React	TypeScript	webpack	$\sum$	$\sum_{\text{all devs}}$
Down → up	2/1	7/–	4/–	4/–	1/–	7/–	3/–	4/1	3/–	6/–	41/2	43
Constant	–/6	–/9	1/6	–/7	1/7	–/7	1/4	–/7	4/7	1/8	8/68	76
Down	3/1	–/–	2/–	5/–	7/–	2/–	4/–	3/–	1/–	1/–	28/1	29
Up	–/–	1/–	2/–	–/–	–/–	1/–	2/–	2/–	–/1	–/–	8/1	9
Horizontal	–/1	–/1	–/3	–/2	–/2	–/2	–/3	–/2	–/2	–/–	–/18	18
Down → up → down	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–
Up → down	–/–	–/–	–/–	1/–	–/–	–/–	–/–	–/–	–/–	–/–	1/–	1
Up → down → up	–/–	–/–	1/–	–/–	1/–	–/–	–/–	–/–	–/–	–/–	2/–	2
Other	5/1	2/–	–/1	–/1	–/1	–/1	–/3	1/–	2/–	2/2	12/10	22

For the issue networks, we end up in largely similar pattern occurrences (see Table 5.3), but we obtain much more occurrences of pattern "constant" for the randomly selected developers (68 out of 100) than we do for the randomly selected developers from the mail networks (35 out of 99).

Answer to RQ$_2$: We observed patterns of transitions regarding developers' trajectory through positions in the organizational structure. In the two most frequent patterns, developers move from the non-hierarchical to the hierarchical part. In one, they move back, and in the other, they stay. Other transitions tend to be rare.

Table 5.4: Descriptive statistics for the 10 most active and 10 randomly selected developers: Commit count, event count, number of active time ranges (i.e., overlapping 6-months ranges within the period denoted in Table 5.1 in which the developer contributed to the mailing list or commented on an issue), and number of time ranges in which the developer was part of the hierarchical part. For each of the two groups of developers, we report the minimum value and the maximum value per project as well as the 25%, 50%, and 75% quantiles.

Project	**Commit Count**										**Event Count**[1]									
	Most active developers					Random developers					Most active developers					Random developers				
	min	max	25%	50%	75%	min	max	25%	50%	75%	min	max	25%	50%	75%	min	max	25%	50%	75%
Django	34	1 231	209	462	1 051	0	3	0	0	0	482	2 895	682	995	1 438	1	47	6	13	23
FFmpeg	0	13 339	19	592	1 093	0	215	0	1	3	1 061	47 411	2 166	5 429	6 990	44	1 064	9	26	37
GCC	0	0	0	0	0	0	0	0	0	0	2 425	18 070	3 158	6 100	13 998	11	568	17	46	193
Git	0	1 674	35	365	792	0	6	0	0	1	2 695	55 548	4 149	6 469	12 096	1	76	15	30	58
LLVM	0	1 558	557	775	917	0	195	0	6	24	2 862	28 885	3 146	4 060	7 061	15	337	27	43	118
ownCloud	0	2 452	1	77	374	0	1	0	0	0	213	874	252	420	468	5	75	9	20	34
QEMU	13	1 891	325	464	659	0	182	0	1	10	2 839	23 188	7 049	8 874	12 751	4	946	24	31	213
Qt	0	647	1	14	155	0	193	0	0	3	229	5 157	371	466	620	5	144	15	20	55
U-Boot	5	2 723	121	392	811	0	87	2	10	28	1 517	28 549	4 597	5 792	16 803	6	594	30	90	148
Wine	81	10 928	546	1 663	2 528	0	761	0	0	27	1 098	7 771	1 354	1 667	2 870	5	398	13	57	162
Angular	0	785	9	78	460	0	0	0	0	0	1 245	13 078	1 877	3 227	8 096	1	16	1	3	9
Atom	0	3 657	45	100	983	0	0	0	0	0	1 307	5 558	2 174	3 004	3 714	1	23	1	2	3
Bootstrap	3	437	22	168	314	0	0	0	0	0	350	19 279	1 943	3 477	7 087	1	17	3	5	7
Electron	0	3 915	27	539	689	0	9	0	0	0	907	10 224	2 224	3 759	5 904	1	38	1	4	9
Flutter	0	1 317	0	7	220	0	0	0	0	0	2 817	28 958	4 008	6 271	9 945	1	29	1	4	16
Moby	0	850	91	222	474	0	2	0	0	0	2 294	46 674	4 312	6 793	8 653	1	28	1	5	13
Node.js	70	1 391	128	336	890	0	1	0	0	0	5 169	25 990	7 732	8 590	18 114	1	125	1	10	19
React	13	941	54	72	545	0	0	0	0	0	929	19 158	2 771	3 954	5 921	1	71	1	3	7
TypeScript	0	2 223	113	1 012	1 194	0	0	0	0	0	1 686	21 174	2 424	7 778	10 199	1	23	1	4	12
webpack	0	3 275	3	38	123	0	0	0	0	0	274	12 935	552	1 099	1 795	1	23	2	5	11

Project	**Active Time Ranges**										**Time Ranges in Hierarchical Part**									
	Most active developers					Random developers					Most active developers					Random developers				
	min	max	25%	50%	75%	min	max	25%	50%	75%	min	max	25%	50%	75%	min	max	25%	50%	75%
Django	19	49	29	44	47	2	24	3	5	8	15	45	28	33	40	0	5	1	2	4
FFmpeg	22	60	30	39	52	2	28	3	8	19	19	60	28	33	42	1	15	1	2	6
GCC	54	69	61	66	69	3	65	8	9	38	23	69	46	55	59	0	6	0	0	2
Git	32	52	41	48	51	2	19	8	12	15	26	52	33	43	45	0	5	0	1	4
LLVM	28	59	34	41	50	5	20	6	13	16	22	47	28	35	41	0	4	0	1	2
ownCloud	8	26	14	17	18	3	13	4	9	11	7	23	13	15	16	1	8	2	4	7
QEMU	25	40	29	37	38	2	39	7	9	19	21	36	25	29	33	0	9	0	1	3
Qt	20	28	26	28	28	4	19	10	12	15	15	27	18	22	26	0	7	0	0	3
U-Boot	29	71	34	39	52	6	34	9	15	18	21	70	28	30	45	0	13	1	3	5
Wine	14	61	41	48	60	4	34	6	10	14	12	59	30	40	54	0	16	1	3	6
Angular	6	25	18	20	24	2	10	2	2	5	5	25	14	17	23	0	2	0	0	1
Atom	12	31	14	19	23	2	15	2	2	4	10	26	13	16	21	0	2	0	0	1
Bootstrap	6	37	15	23	30	1	7	2	4	5	3	37	12	19	25	0	3	0	1	2
Electron	14	30	15	20	26	2	13	2	2	5	12	30	14	17	24	0	5	0	0	1
Flutter	4	22	7	10	15	2	8	2	2	5	4	20	4	5	12	0	1	0	0	1
Moby	18	31	20	28	28	2	11	2	3	7	9	31	18	23	28	0	3	0	1	2
Node.js	18	21	21	21	21	2	13	2	4	7	17	21	19	21	21	0	4	0	0	2
React	14	30	22	26	28	2	10	2	3	4	9	27	14	17	22	0	3	0	0	2
TypeScript	11	25	21	23	25	2	11	2	3	5	11	25	18	22	24	0	2	0	1	2
webpack	7	34	13	18	21	2	9	2	3	7	5	33	7	12	15	0	2	0	0	0

[1] E-mail count (upper 10 projects) or count of issue comments (lower 10 projects), respectively

Discussion The high number of occurrences for pattern "constant" for the randomly selected developers in the issue networks might be caused by the much higher number of project "users" participating in issue discussions only for a short time period, compared to the mail networks (see Table 5.1). When neglecting the pattern "constant" for the randomly chosen developers in issue networks, the most frequent pattern among all developers (most active and randomly chosen developers) is that developers enter the project with only few contacts (pattern "down → up"). Over time, the number of interaction partners rises and the developer climbs the project's hierarchy. This may be indicative of their role changing and gaining coordination responsibilities. Then, the number of interactions decreases again and the developer returns to a small number of contacts. The second-most frequent pattern is similar to the first one in that non-hierarchical developers move to the hierarchical structure's upper regions—however, either they stay or we run out of data before we see them leaving (pattern "down"). This might be caused by the much higher number of project "users" participating in issue discussions only for a short time period (see Table 5.1).

Developers starting in the hierarchical part and moving to the non-hierarchical part are often founders or leaders of the project, who then stopped contributing. We confirmed this for each project that had mailing lists by consulting its website (e.g., for Git, Linus Torvalds is listed as founder; over time, he moved to the non-hierarchical part).

> **Hypothesis 1:** Developers who move up in the hierarchy tend to take more coordination tasks.

5.4.4 Developers' Neighborhood

To address RQ_3, we investigate for individual developers how their neighborhood is distributed over the hierarchical structure.

Results We illustrate an example in Figure 5.10 and summarize the results for all projects in Table 5.5 for mail networks and in Table 5.6 for issue networks. First, we explore the most active developers in a project who have a static neighborhood. These developers interact during their entire life cycle with developers of the hierarchical part (2 out of 100 developers in mail networks, pattern "hierarchical part") or both (61 out of 100 in mail networks, pattern "both"). Second, the most active developers' neighborhood may change, too, which happens in two ways: Either a developer starts their career with contacts mainly from the hierarchical part, and then they interact with developers from both parts (17 out of 100 in mail networks, pattern "hierarchical part → both"), or they start with contacts from both parts and then restrict their interaction to developers of the hierarchical part (19 out of 100 in mail networks, pattern "both → hierarchical part"). Issue networks exhibit similar patterns as described for mail networks.

We also evaluate the neighborhood of randomly selected developers. Their neighborhoods are more stable. These developers interact during their entire life cycle with developers of the hierarchical part (47 out of 99 in mail networks, pattern "hierarchical part") or both parts (34 out of 99 in mail networks, pattern "both"). Only 18 out of 99 randomly selected developers

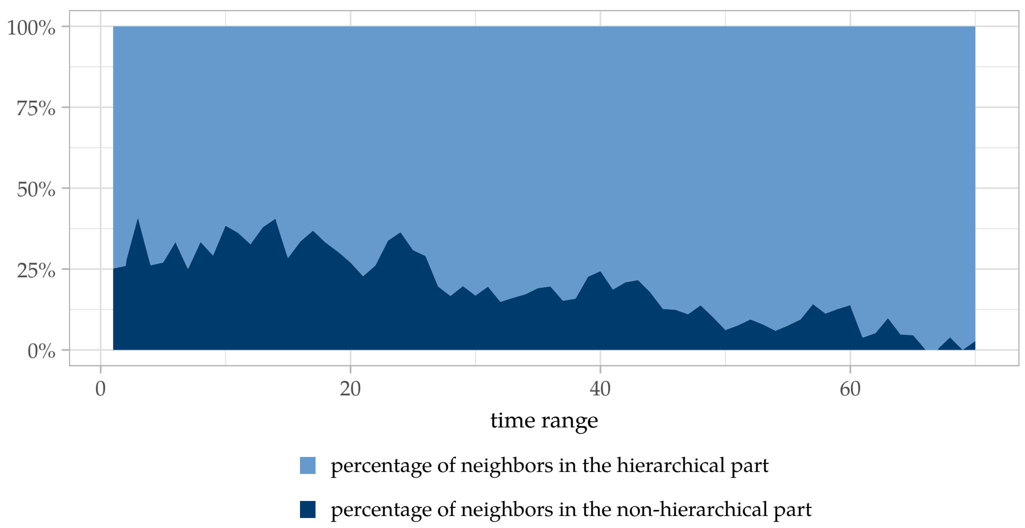


Figure 5.10: Neighborhood of a developer (developer 7507) of the project U-Boot over time. That is, the percentage of the developer's contacts that are from the hierarchical part (light blue) or from the non-hierarchical part (dark blue) at the respective time range.

in mail networks have a dynamic neighborhood. For issue networks, similarly, only 10 out of 100 randomly selected developers have a dynamic neighborhood.

Answer to RQ$_3$: The 10 most active developers interact (via e-mails or issues) with developers of the hierarchical part and sometimes, additionally, with developers of the non-hierarchical part, but rarely exclusively with developers of the non-hierarchical part. This also holds for the randomly selected developers, who mostly interact with developers of only the hierarchical part.

Discussion If a project is assumed to have proficient leadership (which cannot be guaranteed for every project), then it is not unexpected that randomly selected developers of the non-hierarchical part mostly interact with developers of the hierarchical part or both groups, but not solely with developers of the non-hierarchical part: In any discussion, a developer of the hierarchical part can join to add clarifications or to make a decision, which is not unlikely given the role of the developers of the hierarchical part. Consequently, developers of the non-hierarchical part are expected to interact with developers of the hierarchical part in projects that have a well-functioning leadership. Also, it is worth mentioning that the developers of the non-hierarchical part do not deliberately choose their interaction partners, as they cannot influence who is replying to their messages. Developers from the hierarchical part, however, take the role of maintainers and, most likely, decide which discussions they reply to.

Especially the dynamic patterns of the most active developers' neighborhoods are interesting, as these shed light on on- and off-boarding processes. During on-boarding, developers start with interactions from the hierarchical part, and later extend their interaction to developers of both parts. This dynamic might suggest that, when developers enter a project, they start accumulating knowledge from developers of the hierarchical part and only later transfer knowledge to the non-hierarchical part. During off-boarding, we observe that developers focus their interaction to developers of the hierarchical part. Interaction with the

Table 5.5: Frequency of the position of neighborhood contacts in the hierarchy for each subject project for mail networks for the 10 most active/10 randomly selected developers.

Neighborhood Pattern	Django	FFmpeg	GCC	Git	LLVM	ownCloud[1]	QEMU	Qt	U-Boot	Wine	$\sum$	$\sum_{\text{all devs}}$
Static:												
Both	5/2	6/4	5/3	5/4	7/5	6/3	4/3	9/3	5/3	9/4	61/34	95
Hierarchy	–/8	–/4	1/6	–/5	–/3	–/4	–/3	1/4	–/6	–/4	2/47	49
Dynamic:												
Both → hierarchy	4/–	2/1	3/1	1/–	–/–	4/1	4/4	–/1	1/1	–/2	19/11	30
Hierarchy → both	1/–	2/1	1/–	4/1	3/2	–/1	2/–	–/2	4/–	1/–	17/7	34

[1] There are only 9 randomly selected developers in ownCloud.

Table 5.6: Frequency of the position of neighborhood contacts in the hierarchy for each subject project for issue networks for the 10 most active/10 randomly selected developers.

Neighborhood Pattern	Angular	Atom	Bootstrap	Electron	Flutter	Moby	Node.js	React	TypeScript	webpack	$\sum$	$\sum_{\text{all devs}}$
Static:												
Both	6/1	7/1	7/1	10/2	7/1	6/-	10/-	8/3	9/3	6/2	76/14	90
Hierarchy	–/8	–/9	–/7	–/6	–/9	3/8	–/9	–/6	–/6	–/8	3/76	79
Dynamic:												
Both → hierarchy	2/–	3/–	3/1	–/–	2/–	1/2	–/–	1/1	–/1	4/–	16/5	21
Hierarchy → both	2/1	–/–	–/1	–/2	1/–	–/–	–/1	1/–	1/–	–/–	5/5	10

non-hierarchical part or, more specifically, newcomers seems to be present, though. This finding suggests that, when central developers leave, they focus on bringing their ongoing topics to an end, but avoid opening new ones.

Hypothesis 2: When developers enter a project, they start accumulating knowledge from hierarchical developers and only later transfer knowledge to non-hierarchical developers.

Hypothesis 3: Developers who will leave the project do this step-by-step, finishing their ongoing tasks and answering questions to avoid knowledge loss.

5.4.5 Tenure and Programming Activity

Finally, to address RQ_4, we have a look at tenure and programming activity of developers with respect to their position in the hierarchy.

Results In Figure 5.11, we show the developers' tenure and their position in the hierarchy for LLVM: The upper plot encodes tenure in terms of size and color (larger and lighter dots denote shorter tenure values); the middle plot compares the distributions of tenure values of developers in the hierarchy with developers outside the hierarchy; the lower plot shows the progression of average tenure values over time. Overall, the developers in LLVM's hierarchy have, on average, higher tenure values than the developers outside the hierarchy ($p \ll 0.001$; $d = 0.39$). This difference in tenure between developers inside and outside the hierarchy is consistent across all projects that use a mailing list ($p \ll 0.001$; $0.25 \leq d \leq 0.52$). Interestingly, the difference between tenure values of hierarchical and non-hierarchical developers increases over time. Remarkably, also this is consistent across all projects that use a mailing list, except for Qt, where the difference stays constant over time.

For issue-based projects, we obtain slightly different results: Developers in the hierarchy have, on average, higher tenure values than the developers outside the hierarchy. This holds for all projects. However, only for projects Flutter, Node.js, and TypeScript, this difference ($p \ll 0.001$; $0.23 \leq d \leq 0.42$) has a similar effect size than in the projects that use mailing lists. For these three projects, also the difference between the tenure values increases over time, as we already have identified for the projects that use mailing lists. For the remaining seven issue-based projects, the difference in tenure between developers inside and outside the hierarchy still is significant, but with a smaller effect size ($p \ll 0.001$; $0.10 \leq d \leq 0.18$) and without notable patterns over time.

Much like for tenure, we show the results for programming activity for developers of LLVM in Figure 5.12. Developers in the hierarchical part edit most files ($p \ll 0.001$; $d = 0.40$). This difference in programming activity remains existent over time but is fluctuating with regard to its extent. For most projects, we find that, overall, the number of edited files of the non-hierarchical developers is significantly lower than the number of edited files of the hierarchical part. Only for GCC we cannot find any significant difference between developers inside and outside the hierarchy. As already seen for tenure, the difference between the number of edited files of developers inside and outside the hierarchy has a stronger effect on projects that use mailing lists (and project Node.js) ($p \ll 0.001$; $0.12 \leq d \leq 0.47$) than on projects that use the GitHub issue tracker ($p \ll 0.001$; $0.03 \leq d \leq 0.10$).

The dynamics of the individual projects show different patterns, though, which we group into four categories: In the first category, the upper hierarchical part contains developers who edit many files as well as developers who edit no files or only a few files (Angular, Atom, Django, FFmpeg, Flutter, Git, LLVM, ownCloud, React, TypeScript, U-Boot, and Wine). In the second category, the distribution of the number of files edited is split between the hierarchical and the non-hierarchical part: The hierarchical part contains the developers who edit many files, whereas the non-hierarchical parts contains the developers who edit only few files (Bootstrap, Electron, Moby, Node.js, and webpack). In the third category, the pattern is dynamic (QEMU and Qt). For example, in early phases of QEMU, mainly developers of the non-hierarchical part edited files. In later phases, most files were edited by developers of the

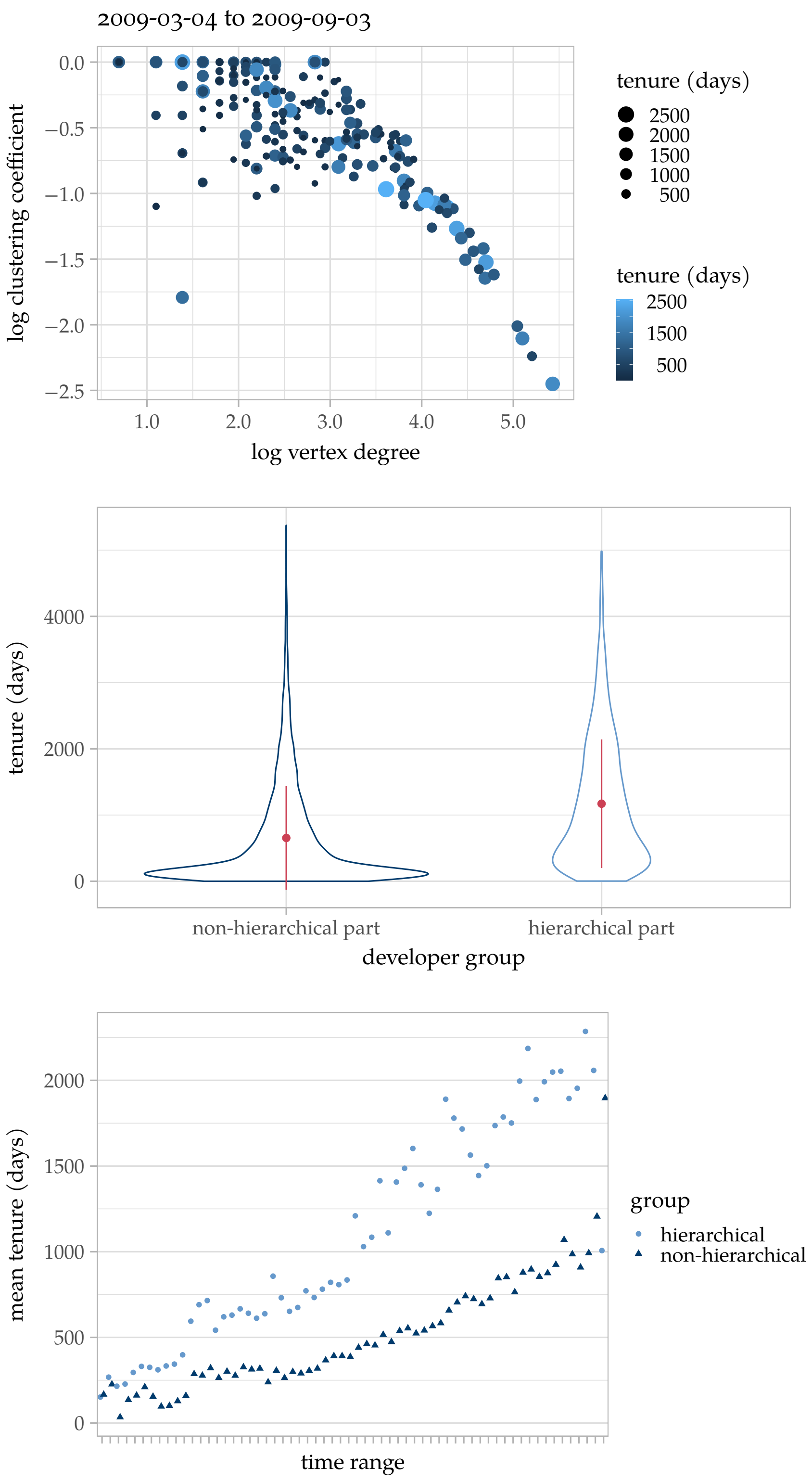


Figure 5.11: Top: tenure of the active developers of LLVM (03-2009 to 09-2009). The bigger and lighter a dot, the longer the developer is already active in the project; middle: distribution of tenure values per group; bottom: average tenure over time.

hierarchical part. In the fourth category is only GCC, for which the developers who edit the files are mainly located in the non-hierarchical part of the network.

Interestingly, for several projects (e.g., GCC, Git, ownCloud, U-Boot, and Wine), the number of developers who program a lot is very low: A maximum number of five developers are responsible for most of the changes. These developers often have a relatively low vertex degree. Furthermore, for some projects and time ranges, we found that developers of the hierarchical part have very few edited files and mainly communicate (GCC, Git, and Qt).

Answer to RQ$_4$: We found a relationship between developers' tenure and their position in the hierarchy: Developers in the hierarchy have a longer tenure than developers outside the hierarchy. Regarding programming activity, developers of the hierarchical part perform most file edits. For some projects, this changes over time: In early project phases, mainly developers of the non-hierarchical part or both parts edit files; later, only developers of the hierarchical part edit files.

Discussion Our data suggest that developers in the hierarchy stay longer in the project. The patterns are consistent with a system where gaining experience through consistent involvement is important for advancement of responsibilities and influence. This finding is interesting in the light of the conjecture that hierarchy reflects role stratification, since developers with the behavior of a core developer consistently appear within the hierarchical part and not in the non-hierarchical part.

Hypothesis 4:	Developers move quickly up the hierarchy when they take on coordination tasks early.
Hypothesis 5:	Consistent contribution, coordination with other project members, and knowledge are important to role advancement in OSS projects.

The number of edited files seems to affect the position in the hierarchy more than the developer's tenure: The more files a developer edits, the more embedded they appear to be in the hierarchy, probably because a higher number of edited files increases the probability that their activity affects many other developers. The interesting cases are when the number of edited files and the position in the hierarchy are unrelated. This could be an indicator for a modular project structure, in which developers of the non-hierarchical part edit files of a certain part of the project, whereas the files that developers of the hierarchical part edit are scattered across many parts of the project. In project GCC, which is an outlier with respect to the programming activity, there could also be an additional explanation for why the developers who edit the files are mainly located in the non-hierarchical part: As GCC is a rather low-level, technical project, which is dependent on technical features that rely on a certain hardware support [HDB18], developers from different hardware manufacturers may add their specific hardware support to the code base, being in the non-hierarchical part of the project communication although accounting for many file edits. Table 5.4 shows that none of the 10 most active developers on GCC's mailing list, who often are also in the hierarchical part, contributed any commit to the source code. This indicates that these most active developers on the mailing list take rather organizational coordination tasks than programming tasks in GCC, which is why the developers who edit files in this project are mainly from the non-hierarchical part of the mail network.

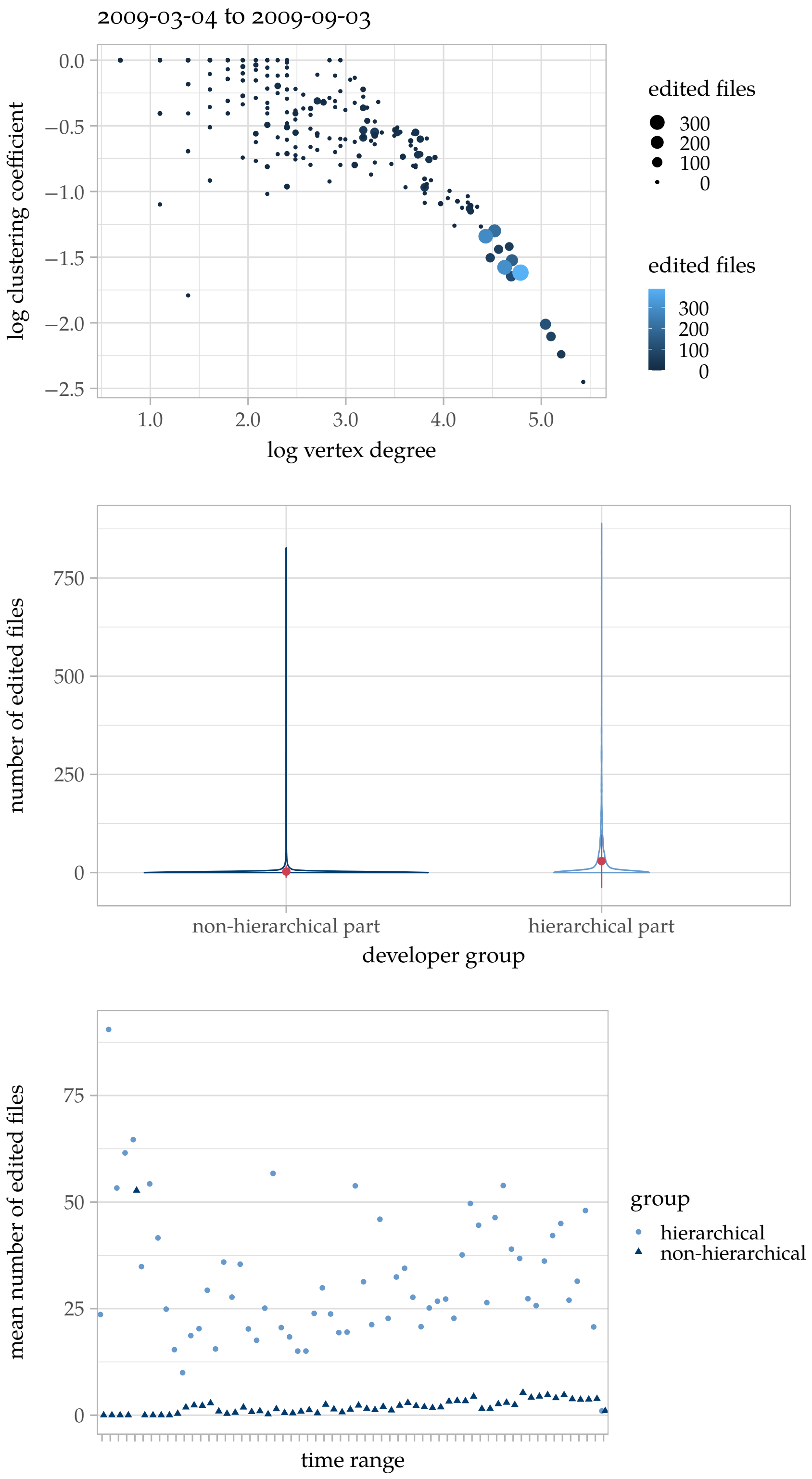


Figure 5.12: Top: number edited files of the active developers of LLVM (03-2009 to 09-2009). The bigger and lighter a dot, the more files the developer has edited; middle: distribution of the number of edited files per group; bottom: average number of edited files over time.

The relationship between developers' positions in the hierarchy and programming activity or tenure occurs to be less pronounced in issue-based projects than in projects that use mailing lists. One possible reason for that could be that there are lots of "users" of issue-based projects who participate in discussions for a large amount of time (e.g., reporting bugs, etc.), which also is represented in the sheer number of participants in the discussions (see Table 5.1). By contrast, there are many developers in the hierarchical part who perform merely project maintenance and pull-request reviews and, therefore, edit only a smaller number of files. Nevertheless, even if the effect is lower for issue-based projects, both kinds of projects have in common that the number of edited files and tenure are higher for developers inside the hierarchical part than outside the hierarchical part.

The fact that the relationship of the number of edited files with the hierarchical part is subject to change speaks in favor of a strong relationship between temporal focus and social contacts. At times when active developers are in the non-hierarchical part, a rather discussion-based group structure seems to establish. At times when the most active developers are at the top of the project's organizational structure, operational activity seems to be the main focus.

Hypothesis 6: When the most active developers are in the non-hierarchical part, they primarily participate in detailed, technically focused discussions with a specific group of developers. Whereas, when the most active developers are at the top of the hierarchy, they primarily take on coordination-related tasks and perform operational maintenance.

5.5 Perspectives

Overall, we found that the organizational structure co-evolves with OSS projects, despite the absence of external pressure to form any specific kind of structure. Specifically, it (a) regularly splits into a hierarchical and a non-hierarchical part, consistent with anecdotal evidence first seen by Joblin et al. [JAM17]. Over time, the number of developers increases until reaching a peak, followed by a decrease, which is accompanied by a restructuring process. Growth in the number of developers usually leads to a decrease in the portion of developers who are organized hierarchically. From the perspective of individual developers, (b) the on-boarding of the most active developers follows typical patterns. We could confirm the belief that developers often start loosely connected in the non-hierarchical part [SSG14], then tightly integrate into the hierarchical part, until they move to the non-hierarchical part again and most likely leave the project. During this process, (c) developers' coding activity and their tenure can drive their hierarchical position. That is, on the one hand, early pioneers of a project are as likely to be in the upper ranges of the hierarchy as new members of a project. On the other hand, developer roles can adapt flexibly to changing project situations. This is relevant for research on success factors for climbing the social ladder of OSS projects [e.g., HHB+18; TGW+22] and helps to better understand the social dynamics integrating project newcomers. Padhye et al. [PMS14] found that core developers are open to accept bug fixes and documentation changes from peripheral developers, but not to proposed feature enhancements from peripheral developers. This habit appears inefficient—our method, when combined with a content analysis on the kind of contribution by the developers of the two parts, can help to identify such *community smells* [PT21; TKL+15; TPK19].

Second, also at the group level, we observed considerable flexibility with regard to the number of groups in which developers want to engage. The prevalent hybrid structure that we observed contains developers whose clustering coefficient and vertex degree do not match the rules of hierarchical organization. Combining our approach with dynamic group analysis [e.g., HAT+21] might provide insights into the hub function of central developers. For example, we observed that many former top developers leave the project with a high clustering coefficient, that is, their last interaction partners were from a well-connected subgroup. Further research on these groups shall complement analyses of developer tenure and knowledge conservation [e.g., RCO19]. Moreover, our method can be used to further investigate the role of developers within stable or dynamic subgroups of the project (see Chapter 4), with respect to their position in the hierarchy to get more insights into how developers in the hierarchical part are connected to the different subgroups, and to investigate their role with respect to programming activity and communication activity. This might provide further insights into group dynamics and may be used to derive recommendations for project managers on how to reduce developer turnover, knowledge loss, and maintain a project successfully. There is already evidence that socio-technical factors derived from network representations contain highly relevant information about the future success of a project [JA22]. The patterns found in our study are useful for providing additional context for predictive models to increase their effectiveness. Currently, predictive models lack the temporally rich patterns that we have identified. We believe that a coherent treatment of the temporal dimension is likely necessary to move beyond predictive analytics to achieve the end goal of prescriptive analytics.

Third, we found that structural changes, such as losing a substantial number of developers, manifest in the hierarchy directly. At times of change, developers who do not edit many files in the respective time range dominate the hierarchical structure. An interesting case is ownCloud, where we observed a breakdown of the hierarchical structure 2 years before the fork of Nextcloud, only emerging again afterwards. This possibly hints at the organizational disturbances that eventually led to the fork. We will have a closer look at such disturbances in Chapter 7. These insights illustrate that our analysis can add to the methodological toolbox for research in change management. Since our study focused on popular projects, which may be a proxy for success, a promising future direction is to explore these structural changes for projects that became deprecated.

5.6 Threats to Validity

As always in empirical studies, and especially due to the exploratory nature of our study, the validity of our results may be threatened. Therefore, we have grouped the potential threats into two different categories and discuss them in the following.

External Validity We selected 20 OSS projects with a wide range of domains, programming languages, and sizes (see Table 5.1), so our results appear generalizable to similar projects. For generalization to different communication channels, we investigated projects that use mailing lists as their main communication channel as well as projects that use an issue tracker for communication, and we ended up in similar results for both types of projects. Clearly, 20 projects cannot cover the diversity of OSS projects out there, and our study is naturally not meant to generalize to any OSS project. Still, some patterns and trends that we found are surprisingly pronounced and consistent across our subject projects. This demonstrates the fundamental power of our method to reveal such complex patterns and to study them in depth.

Another threat arises from our selection of only the 10 most active developers and 10 further random developers for analyzing how individual developer behavior affects the position in the hierarchy. However, since the 10 most active developers cover a considerable amount of interaction, our results are, nevertheless, relevant for the project as a whole.

Construct Validity The fact that we set the parameters of our classification algorithm for decomposing developer networks into a hierarchical and a non-hierarchical part only with a manually labeled sample threatens construct validity. To gain further confidence, we triangulated our classification with the dichotomous core–peripheral classification of Joblin et al. [JAM17], which relies on eigenvector centrality. We found that the two classifications are consistent: In most projects, the set of core developers and the set of developers in the hierarchical part overlap to a large extent, but the core developers' number is often smaller than the number of developers in the hierarchical part. Core developers are usually in the hierarchical part, but their percentage usually decreases as the project evolves.

A further threat to construct validity is the way we build our networks. As mailing lists or issue discussions, respectively, are the predominant communication channels in our subject projects, there was no need to consider further communication channels, such as chat systems or online forums. To learn about the relative influence of different types of developer interaction, we constructed not only pure *issue* or *mail* networks, respectively, but also combined *cochange+issue* or *cochange+mail* networks, respectively, which consist of communication edges and cochange interactions based on contributions to a common file (see Section 2.3.1). This way, we obtain a more comprehensive view on the community, since we include more information about the project than only considering activities on a single communication channel. Using the combined networks that consist of information from two interaction channels, we found that these networks receive their structural properties mainly from the communication activities (which are much more in number). Overall, we obtain the same results, regardless of whether we use pure communication networks or the combined networks that consist of communication and cochange information. We provide the results using these combined networks that convey communication and cochange information for all our research questions in a separate section on our supplementary website.[6]

Another threat is the definition of tenure: We do not explicitly consider extended phases of developers' inactivity, during which developers do not contribute to the project. The tenure's end date is always the end of the time range, since we assume the developer's ongoing activeness. Our definition of programming activity also poses a threat: Defining it as the number of edited files may be too simple, because the changes can be of different size (in terms of LOC) and of different complexity. Giving weights to the changes accordingly would make our analysis much more complex, though. Zhou and Mockus [ZM10] describe another method based on edited files that could improve a follow-up study.

6 https://hierarchypaper.bitbucket.io/public/#results-by-research-questions-combined-networks-with-e-mailissue-communication-and-co-change

5.7 Conclusion

Prior work indicated that, over time, highly successful OSS projects develop a *hybrid* organizational structure that comprises a *hierarchical* part and a *non-hierarchical* part. To study this phenomenon systematically, we conducted a longitudinal study of 20 popular OSS projects. In particular, we searched for structural patterns with respect to hierarchy in OSS projects. Therefore, we developed an automated method to determine the hierarchical and the non-hierarchical part in developer networks. We tracked the neighborhood and movement of individual developers to understand their placement and trajectory through the hierarchy, and we analyzed the relationships of the developers' organizational position with their tenure and programming activity.

Most notably, we found that, with an increasing number of developers in our subject projects, the portion of developers in the hierarchical part decreases to as little as ~10%, in some projects even less than that. Essentially, in almost all cases, a hybrid organizational structure emerged that consists of a hierarchical part and a non-hierarchical part, independent of project size, domain, or used communication channel. A deeper investigation lets us link network hierarchy more closely to actual developer behavior: Very active developers are well connected in the hierarchy, starting during their on-boarding phase, and stabilizing this connection over time. Surprisingly, tenure is also associated with hierarchical position—any developer can quickly and flexibly take over responsibility in the projects, a promise of OSS development. The high correlation between programming activity and hierarchical position confirms that, often, OSS developers have a dual role for contributing code and taking over coordination efforts.

Whereas early pioneers of a project are likely to be in the upper ranges of the hierarchy, developer roles adapt flexibly to changing project situations. The organizational structure of OSS projects is subject to constant change, which manifests itself in developer turnover and changing developer roles, and therefore provides the possibility to newcomers to climb up in the organizational hierarchy with increasing tenure and increasing project involvement. Our method can be used to gain further insights into structural changes in project organization and hierarchy, to identify potential organizational community smells, and, eventually, to develop countermeasures against potential knowledge loss in OSS projects (e.g., when a core developer in the top of the hierarchy is moving to the non-hierarchical part and is potentially about to leave).

In summary, our study (a) provides a methodological basis for further investigations of hierarchy formation, (b) suggests a number of hypotheses on prevalent organizational patterns and trends in OSS projects to be addressed in future work, and (c) may ultimately help to guide the governance of organizational structures.

6 Identifying Core Developers in Open-Source Software Projects

This chapter shares material with Bock et al. [BAJ+23].

Coordination among software developers is critical to ensure software quality and to drive software evolution [CH13; HMR06; KS95; KSD11]. This holds especially for open-source software (OSS) projects, in which volunteers can participate and developers are globally distributed, often not knowing each other personally [Bir11; GHP99]. With increasing project size and popularity, developers who hold leadership roles (i.e., who take care of the project's health and are highly involved in its long-term maintenance) are crucial for a thriving evolution of the project [CLL+17; Lon06; XJS09; YHH04]. As opposed to closed-source software projects, where leadership positions are typically established by mandated organization structures and well-determined within the organization, OSS projects often do not provide explicit, publicly recognizable information regarding group structures and project responsibilities. Still, in OSS projects, hierarchical group structures among developers do exist [see Chapter 5; TLV13; YK03; ZCM+17], but they are typically not determined by a centralized authority but rather emerge following principles of self-organization [CLW+07; Yu08]. So, despite the lack of predefined hierarchical group structures in OSS projects, certain developers take particular responsibilities in the project, such as performing maintenance tasks or implementing core functionality. Such developers, often called *maintainers* or *core developers* [JAH+17; JSW11; MFH02; NYN+02], become crucial to these projects, as they hold key knowledge about source code and software architecture [RZD+16] and shape the character and culture of the project [AKH+16; WFW+20; ZM12], especially when OSS projects grow fast in terms of developers and amount of source code [EZZ19]. As a prominent example, the Linux kernel maintains a public list of maintainers, who are responsible for specific subsystems and who have deep knowledge about the project [ZCM+17]. Using this list, external and internal developers can find out whom to contact for any matters. Unfortunately, many OSS projects do not maintain such a public, curated list. Especially for newcomers in a software project, but also for developers that are not consistently active (also called *peripheral developers* [CH05; JAH+17; MFH02; NYN+02]), it is important to know who is playing the role of a core developer, who is making decisions and maintaining the code base, and who will finally accept or reject code contributions [LS07; STG19]. In well-organized projects, this might not be an actual problem for newcomers, as one can easily check who has recently reviewed or merged pull requests. Some projects even use bots to automatically assign developers for reviewing a new pull request [WSS+18]. However, in projects that are badly organized and which do not use such well-structured code-review processes, it might be not that easy to find out whom to contact in the case of questions, for instance. Knowing how developers organize is also important for companies, to decide whether and how to invest in a project or how to efficiently contribute to an OSS project [BGL+21; DSP18; ZMM+16; ZZM+19]. For researchers, knowing about core developers also aids the investigation of a project's hierarchical structure, which helps avoid high developer turnover and identify organizational smells that could endanger project success [CDO+12; CS17; CVS+18; GLM06; KSD11; RCO19; TKF23].

There is a growing corpus of research that attempts to *automatically* extract information about the core developers of a project. A state-of-the-art approach is to construct a developer network from various data sources (e.g., commits, e-mails, issue comments) and to apply network-based centrality metrics. The assumption is that core developers are responsible for about 80% of the contributions and activities in the project [JAH+17; JAM17; JMA+15; LS07; MW11; YMK+15; ZYW11], attaining central positions within the network due to the high number of contributions and many interactions with other developers. While this is an appealing approach, the actual performance of such classification methods on the identification of core developers is largely unclear, especially as there are various ways on how to construct developer networks and also many, possibly contradicting classification methods can be applied thereafter.

Therefore, we devise a method to *automatically* derive a set of core developers and maintainers from privileged events in GitHub issue discussions and pull requests. The rationale is that GitHub permits triggering certain events (e.g., merging a pull request) only to user accounts that have been assigned to a certain role. Nevertheless, these privileged events cannot be used to identify potential candidates for future core developers before they get any privileges in the project and also not for projects that use other social coding platforms than GitHub issues. Therefore, other classification methods for identifying core developers are still essential, which is why it is also our goal to assess the accuracy of the state-of-the-art classification methods. In an empirical study on 25 widely-used and well-known GitHub projects, we (1) validate our automatically derived set of core developers based on privileged issue events with publicly available, project-reported lists of maintainers or core developers for projects that provide such lists, and we (2) use our set of core developers to assess the accuracy of various state-of-the-art unsupervised developer-classification methods based on well-established count-based metrics (e.g., commit count) or network-based metrics (e.g., degree centrality).

Our results indicate that our automatically identified set of core developers is sound and that the accuracy of state-of-the-art unsupervised developer-classification methods depends mainly on the data source (commit data vs. issue data) rather than the network-construction method (e.g., directed vs. undirected). Combining issue data and commit data behaves similarly to just using issue data. Our results hold the potential to help researchers and OSS communities choose the appropriate *unsupervised* classification method for identifying core developers or potential maintainers.

Moreover, our method can help create reliable ground-truth data for training *supervised* classification methods, as—to the best of our knowledge—the lack of sufficient volumes of ground-truth data has prevented supervised learning methods from being developed.

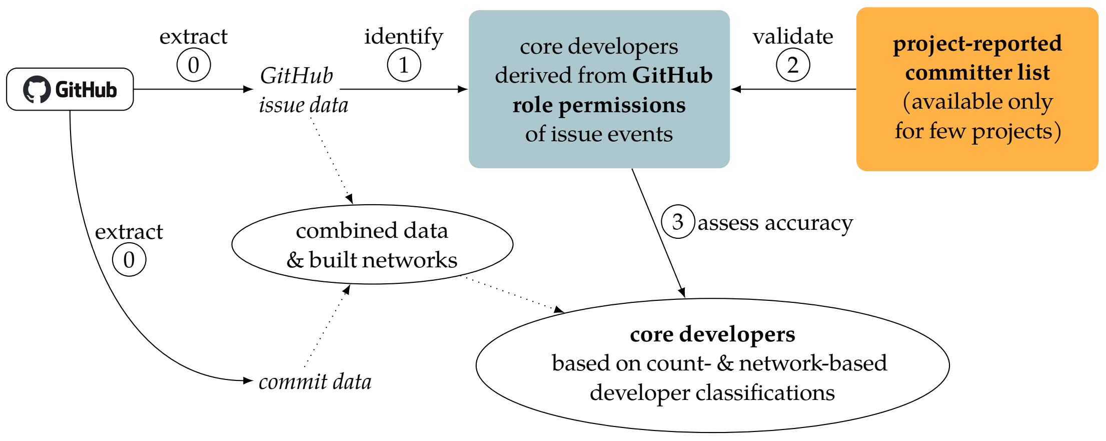

GitHub
extract ⓪
GitHub issue data
identify ①
core developers derived from **GitHub role permissions** of issue events
validate ②
project-reported committer list (available only for few projects)
extract ⓪
combined data & built networks
③ assess accuracy
commit data
core developers based on count- & network-based developer classifications

Figure 6.1: Overview of our approach: (⓪) extract data from GitHub, (①) identify a set of core developers based on the events in issues and pull requests, (②) validate it with project-reported lists, (③) assess the accuracy of state-of-the-art unsupervised classification methods.

Although we devise a method to automatically derive a set of core developers from privileged events in GitHub issues, there is still a need for supervised classification methods: On the one hand, it is more tedious to get the necessary data for the extraction of privileged events than for the state-of-the-art count-based or network-based classification methods, as privileged events may not be available for projects that use other social coding platforms than GitHub. On the other hand, supervised classification can be used to identify potential future core developers and maintainers from activity data before the developers get any privileges in the project. Of course, becoming a core developer might not only be a matter of participation, but also a matter of trust [ABW+13; CLN17; SMS11]. As trust, however, cannot be quantified in the accessible data, using activity data can be a first step toward finding potential candidates, especially since accepted commits and intensive involvements in issue discussions can be used as a first indicator for the trustworthiness of a candidate [DST+12].

In summary, we make the following contributions:

- An overview of literature on how core developers have been identified in previous work (see Tables 6.1 and 6.2).
- A method to *automatically* identify a set of core developers based on role permissions of events triggered in GitHub issues and pull requests (see step ① in Figure 6.1).
- A validation of our automatically identified set of core developers based on privileged issue events with official, publicly available maintainer or committer lists (see step ② in Figure 6.1).
- An assessment of the accuracy of several state-of-the-art *unsupervised* developer-role classification methods that use count- and network-based metrics (see step ③ in Figure 6.1). We found that network-construction methods do not make a substantial difference, whereas the data source plays a more vital role: Classification methods using commit data perform better than classification methods using issue data. Combining issue and commit data behaves similarly to just using issue data, as the combination is dominated by the issue data.

- A small explorative experiment on how our automatically identified set of core developers can be used for training *supervised* classification methods, from which we derive two hypotheses that should serve as a starting point for future research on this topic.
- A replication package including pseudonymized raw data, classification data, tooling, analysis scripts, and results on a supplementary website: `https://se-sic.github.io/paper-developer-classifications/` and `https://zenodo.org/record/7775882`.

6.1 Background & Related Work

In this section, we provide background information and related work on how core developers can be identified, as well as information on GitHub's user permissions, which we use for automatically identifying core developers in our study. For background information on different developer roles, we refer to Section 2.2, in which we have introduced the roles of *core* and *peripheral* developers and have discussed also different role models that have been proposed in the literature. In Chapter 4, we have already looked at groups of developers that form up around latent tasks and determined the importance of developers based on the importance of the latent tasks in which the developers are involved. In what follows, we focus on different methods that have been used in the literature to identify core developers (or, in other words, to classify developers into core and peripheral).

6.1.1 Core-Developer Identification in the Literature

Researchers have developed and investigated various unsupervised, manually tuned methods to identify core developers in OSS projects. To obtain an overview of how core developers have been identified in previous work, we performed a series of searches on multiple different search engines (GoogleScholar, IEEEexplore, ACM Digital Library) using the search terms "core developer" and "identification of core developers", without restricting venue and publication year. We manually checked the papers of the search results and looked for core-developer identification techniques that were described, mentioned, or referenced in these papers. Moreover, we collected such techniques also from papers that have been referenced in these papers in a backward-search fashion. In what follows, we summarize the core-developer identification techniques that we have collected from the literature. Additionally, we manually extracted the used data sources and classification metrics from all the collected papers. In Table 6.1, we provide an overview of the data sources and network types that have been used in the literature to identify core developers; in Table 6.2 we summarize the used metrics.

Mockus et al. [MFH02] proposed that highly active developers (in terms of the number of contributions) should be considered as core developers. They investigated the projects Mozilla and Apache Web Server and identified that only a small number of 10–15 developers are responsible for around 80% of the source-code contributions. Dinh-Trong and Bieman [DB05] replicated their study on FreeBSD and showed that even more than 15 developers contribute 80% of the code base. Follow-up research showed that Zipf's law holds for the number of commits that are authored by a developer and, consequently, the top 20% of the developers are responsible for 80% of the commits [CWL+06]. Using this 80% threshold is a commonly

Table 6.1: Classification data and network types used for core-developer identification in the literature.

	Data / Network Type	Papers
Count-based	Commits	[ARK+18; CBO+20; CVS+18; FSV20; RG06; RGH09; RH07; RZD+16; TRC10; VVH18; YMK+15]
	E-mails	[KJR+16]
	Issues	[CWL+06; TPS+19]
	Commits, e-mails	[JAH+17; KS02; OSG+15; OSO+12]
	Commits, e-mails, issues	[DB05; MFH02]
Network-based	Cochange	[CLL+17; JAM17; LRG+06; PBD+14; PT21]
	E-mail	[BGD+06; JEB+23; PBD+14; SSS17]
	Issue	[AW18; CH05; CWL+06; JEB+23; LM13; LM14; PBD+14]
	Cochange, e-mail	[BGD+06; BGD+07; Bir11; JAH+17; JEB+23; OSG+15; OSO+12; ZYW11]
	Cochange, issue	[BC14; EKB+17; JEB+23; TPS+19]

Table 6.2: Classification metrics used for core-developer identification in the literature.

	Metrics	Papers
Count-based	Commits	[ARK+18; CVS+18; FSV20; JAH+17; OSO+12; RG06; RGH09; RH07; SFD05; TRC10; VVH18; YMK+15]
	Lines of code (LOC)	[JAH+17; KS02; MFH02; RZD+16; YMK+15]
	Authored files	[CBO+20]
	E-mails	[KJR+16]
	Modification requests	[MFH02]
	Issue comments	[CWL+06]
	Pull-request comments	[TPS+19]
Network-based	Degree centrality	[AW18; BC14; BGD+06; Bir11; CH05; CLL+17; EKB+17; JAH+17; LRG+06; OSG+15; PBD+14; SFD05; SSS17; TMB10; ZYW11]
	Betweenness centrality	[AW18; BC14; BGD+06; Bir11; CH05; CLL+17; EKB+17; LRG+06; OSG+15; SSS17; TMB10; ZYW11]
	Eigenvector centrality	[AW18; BC14; EKB+17; JAH+17; JEB+23; OSG+15; OSO+12; SSS17]
	Closeness centrality	[AW18; BC14; CLL+17; EKB+17; LRG+06; OSG+15; SSS17]
	PageRank	[AW18; BC14; EKB+17; ZYW11]
	Clustering coefficient	[JAH+17; JAM17; LRG+06]
	Hierarchy centrality	[JAH+17; JAM17; JEB+23]
	Density	[LM13; LM14]
	Modularity	[BGD+07; JAM17]
	Eccentricity	[AW18]
	Graph partitioning	[CWL+06]
	HITS	[ZYW11]
	Scale-freeness	[JAM17]

used approach when classifying developers based on their commit count [ARK+18; CVS+18; CWL+06; DB05; FSV20; KS02; MFH02; RG06; RGH09; RZD+16; TRC10; YMK+15]. For this reason, we also use this threshold in our validation study for the developer classification based on commit count when we compare our approach with the state-of-the-art classification methods (see Section 6.2.4). Valiev et al. [VVH18] even used a threshold of 90% of contributions per month, which was selected based on empirical observations on the Python Package Index ecosystem. Coelho et al. [CVS+18] and Ferreira et al. [FSV20] extended this approach by adding the additional restriction that a core developer has to have authored, at least, 5% of the total number of commits in the project. Instead of considering the number of commits, Canedo et al. [CBO+20] required a core developer to have authored, at least, 50% of the files in the project. Nonetheless, relying on count-based operationalizations for developer classification provides only limited insights into organizational matters of OSS projects, as relationships among developers (possibly varying over time) are neglected [JAH+17].

To incorporate relationships among developers, Crowston et al. [CWL+06] suggested three different approaches on how to classify developers into core and peripheral: For some of their analyzed projects, they found project-reported lists on the projects' websites, containing formal roles of developers (e.g., who is allowed to check-in source code to the version-control system). This approach is similar to how we validate our set of core developers (see Section 6.2.3). Using project-reported lists to determine core developers has also been adopted by other researches [CS17; KJR+16; WCE+17]. However, such project-reported lists are only available for few projects. In their second approach, Crowston et al. [CWL+06] considered developers to be core if they contribute, at least, one third of the total number of comments on bug-tracker data from SourceForge. This approach is similar to the count-based classification mentioned above and is based on the sheer number of posted comments. In their third approach [CWL+06], they built communication networks on bug reports and applied graph partitioning algorithms, treating the tightly interconnected group as core developers. All three approaches led to different results as the project-reported list seemed to be incomplete and count-based and network-based classification focus on different characteristics of core developers.

Network-based developer classification has been gaining considerable momentum in software-engineering research. De Souza et al. [SFD05] created networks based on developers' contributions to the same modules. Bird et al. [BGD+07] built networks based on mailing-list communication, and they used basic social-network analysis techniques to identify developers' roles [BGD+06; BPD+08; Bir11]. Licorish and MacDonell [LM13; LM14] used networks built from the communication on tasks in IBM's Rational Jazz development environment and considered a developer to be core if they had contributed to the communication of, at least, one third of the tasks. Oliva et al. [OSG+15; OSO+12] identified core developers in developer networks based on mailing lists and version-control systems. They used centrality metrics combined with a quartile analysis to determine core developers, but they investigated only a short time period of a small project. We, thus, investigate a history of several years for 25 projects. Moreover, in our validation study, we build developer networks based on version-control systems and communication in issues, not based on mailing lists.

Joblin et al. [JAH+17] constructed developer networks from the version-control system data and mailing lists of 10 OSS projects. They applied the network centrality metrics degree centrality, eigenvector centrality, and hierarchy centrality, which capture structural differences in the relationships among developers. To evaluate the classification outcomes, they conducted a survey among 166 developers of the examined projects. According to the survey, the network-based developer classifications outperformed the simpler count-based developer classifications. Notably, there is a substantial corpus of work that relies on network-based developer classifications to investigate characteristics of core and peripheral developers and organizational structures in OSS projects [AW18; BC14; CLL+17; EKB+17; JAM17; LRG+06; PBD+14; PT21; SSS17; TMB10; TPS+19]. Naturally, this work depends on the accuracy of the classifications, which is exactly the motivation of our validation study, in which we assess the accuracy of a selection of the above mentioned classification methods.

Zhang et al. [ZYW11] investigated whether the social-network metrics vertex degree (i.e., degree centrality), PageRank [LM06] (a variant of eigenvector centrality), HITS [GKR98] (an algorithm to detect hubs and authorities), and betweenness of vertices in the network are capable of identifying core developers in OSS projects. They assessed the accuracy of their classification methods against a self-created ground truth based on "the right to post messages" on the developer mailing list of the project ArgoUML, considering only developers that had regular contributions to source code and mailing list. All four network metrics performed similarly well in detecting core developers, having a recall of more than 60% and a precision of about 60%. Similarly to their study, we also assess the accuracy of classification metrics using a ground truth that is based on the permissions to perform privileged events on GitHub issues. We improve over their study by investigating a variety of GitHub projects, constructing different types of developer networks, and using network-based centrality metrics that have been used in state-of-the-art research on developer classification.

Closest to our automatic core-developer identification approach, Wang et al. [WFW+20] define the role of an elite developer as a developer having write permission in a GitHub project. They identify elite developers within time ranges of 3 months based on performed tasks that require write permission. This approach is similar to how we identify core developers. However, there are a couple of differences between their approach and our approach: Whereas they access GitHub's event API to gather the performed tasks of a project (e.g., whether somebody has forked a project or starts watching a project) and merge it with additional information from other data sources, we directly access GitHub's issues API, which contains more detailed information on the events that happen specifically in issues and pull requests (e.g., a comment was added, an issue was labeled, a pull-request was merged, etc.). Another difference is that they needed to classify the gathered data into a complex taxonomy of event types (communicative, organizational, typical, supportive), whereas we just use GitHub's official description of all the possible issue event types, which contains information on the user permission that is needed to trigger a specific issue event. Finally, while Wang et al. use fixed time ranges of 3 months, we study the time difference between certain issue events and investigate different time ranges between 3 and 12 months.

6.1.2 User Permissions on GitHub

As we identify core developers based on user permissions, we briefly describe which user roles and permission levels exist on GitHub. First of all, GitHub distinguishes between *organization* (shared account across many projects) and *user* accounts. Repositories (also called projects) can be created from either an organization or a user account. Each organization on GitHub has, at least, one organization owner (which is a normal user). Organization owners and project admins can set the user permission individually for each project and user. At the project level, GitHub users can have one of the following permissions:[1]

- ***Read***: default permission for any user seeing a project, "recommended for non-code contributors who want to view or discuss your project"[1]. Users with read permission can open issues or pull-requests, submit reviews on pull requests, write comments, and close issues or pull requests that have been opened by themselves.
- ***Triage***: "recommended to proactively manage issues and pull requests without write access"[1]. In addition to read permission, a user with triage permission can set labels, close and reopen issues, assign issues to users, and request reviews from users.
- ***Write***: "recommended for contributors who actively push to your project"[1]. In addition to triage permission, a user with write permission can push to a repository (i.e., directly push source code to the repository), merge pull requests, publish releases, and submit reviews that "affect a pull request's mergeability"[1].
- ***Maintain***: "recommended for project managers who need to manage the repository without access to sensitive and destructive actions"[1]. In addition to write permission, a user with maintain permission can protect branches, decide on how pull requests can be merged, and restrict which other users can open issues or pull requests, etc.
- ***Admin***: "recommended for people who need full access to the project, including sensitive and destructive actions like managing security or deleting repository"[1]. When creating a project, the user creating the project has admin permission. Organization owners implicitly have admin permissions on all projects of the organization.

In our study, we make use of these permissions to automatically identify a set of core developers by mapping GitHub's user permissions to our developer roles (see Section 6.2.2). Unfortunately, it is not publicly accessible which user has which permission in a GitHub project. Instead, we need to infer the users' permissions from certain actions GitHub users perform on the issues and pull requests of a specific project.

6.2 Methodology

As illustrated in Figure 6.1 (page 153), our validation study proceeds in four steps: In the initial step (⓪), we extract commit and issue data from GitHub. Then (①), we identify the developers who have triggered a privileged issue event. Following this (②), we validate this set of developers with official, project-reported committer lists. Finally (③), we perform unsupervised developer classifications and assess their accuracy.

6.2.1 Data Extraction

For this study, we extract commit metadata (including author name, e-mail address, author date, and the names of the changed files of a commit) from Git, and we extract issue and pull-request data (including review data, review comments, and all other comments of a pull

1 `https://docs.github.com/en/github/setting-up-and-managing-organizations-and-teams/repository-permission-levels-for-an-organization#repository-access-for-each-permission-level` (accessed at 2020-09-09)

Table 6.3: GitHub issue events and the role permissions needed to be able to trigger them.

Privileged Events (*write*, *maintain*, or *admin* permission)	Extended Events (at least, *triage* permission)	Common Events (at least, *read* permission)
`added_to_project`, `converted_note_to_issue`, `deployed`, `deployment_environment_changed`, `locked`, `merged`, `moved_columns_in_project`, `pinned`, `removed_from_project`, `review_dismissed`, `transferred`, `unlocked`, `unpinned`, `user_blocked`	`assigned`, `demilestoned`, `labeled`, `marked_as_duplicate`, `milestoned`, `unassigned`, `unlabeled`, `unmarked_as_duplicate`	`automatic_base_change_failed`, `automatic_base_change_succeeded`, `base_ref_changed`, `closed`, `comment_deleted`, `commented`, `committed`, `connected`, `convert_to_draft`, `created`, `cross_referenced`, `disconnected`, `head_ref_deleted`, `head_ref_force_pushed`, `head_ref_restored`, `mentioned`, `ready_for_review`, `referenced`, `referenced_by`, `renamed`, `reopened`, `review_request_removed`, `review_requested`, `reviewed`, `subscribed`, `unsubscribed`

request) from GitHub's official REST API[2], using the tools Codeface and GitHubWrapper, which we have introduced in Section 2.4. Afterwards, we combine the data from both sources and perform developer disambiguation as well as bot detection as explained in Section 2.4.

The extracted issue metadata contain information on which user has triggered which event (commented, labeled, merged, etc.), on which issue or pull request, and at which point in time. After bot detection, we remove all bot-triggered events from the extracted GitHub issue data.

6.2.2 Identifying Core Developers Based on Issue Events

To identify core developers, we analyze the events that appear in the above extracted issue data. For each of the events, we look up in GitHub's official event documentation[3] which user permission on GitHub is needed to trigger the event (e.g., write permission is needed to merge a pull request). This way, we form three categories of events: *Common events*, which can be triggered by everyone (e.g., write a comment), *extended events* that can be triggered only when having, at least, triage permission (e.g., apply a label), and *privileged events* that can be triggered only when having, at least, write permission (e.g., merge a pull request). In Table 6.3, we provide an overview of all events and the categories we assigned them to.

As a next step, we analyze which user has triggered which event. Each user who has triggered a privileged event is considered a core developer (at least, in the time range that contains the event). All remaining users are considered peripheral developers. We base this decision on the fact that events that require, at least, write permission are either related to critical tasks for project maintenance (blocking users or locking issues) or to decision making

2 `https://docs.github.com/en/rest/` (accessed at 2022-03-14)
3 GitHub provides a list of possible events here: `https://docs.github.com/en/developers/webhooks-and-events/issue-event-types/` (accessed at 2020-09-09)

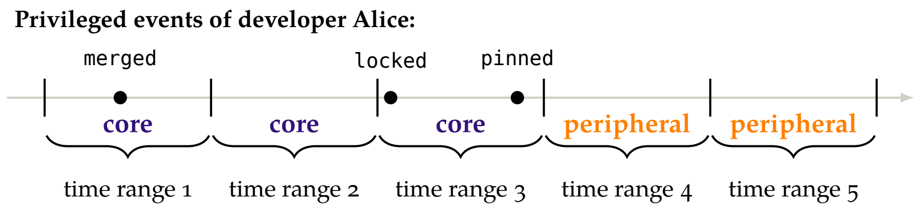


Figure 6.2: Analyzing the issue events of developer Alice: For each time range, check whether Alice has triggered privileged events. If so, Alice is considered core (time ranges 1 and 3). For the remaining ranges, check whether Alice has triggered privileged events in the directly preceding *and* in the directly succeeding time range. If so, then also consider Alice core (time range 2), otherwise peripheral (time ranges 4 and 5).

tasks that require deep project knowledge (e.g., merging or rejecting a pull request), and, therefore, distinguish a core developer.[4]

It is important to note that we cannot know from the data when exactly a developer has received write permission, nor whether or when the write permission may have been revoked. Instead, we only see when the developer happened to trigger an event that requires write permission. So, when considering a developer as core based on the triggered privileged events in a certain time range, we might potentially overlook core developers who only rarely trigger privileged events. To mitigate this risk, we explore how regularly core developers make use of privileged events. This way, we test whether time ranges of 3 to 12 months are long enough—and therefore justified—to identify core developers based on the usage of privileged events within the time range. Therefore, we investigate the following research question:

> **RQ$_1$:** How long is the typical time difference between a developer's events that require, at least, write permission?

As developers might be absent for a particular time range (e.g., due to illness or vacation, etc.), we will exploit a temporal smoothness assumption to improve the accuracy of our method. That is, we treat developers also as core even when they are not triggering privileged events in the current time range but do so in both the previous and succeeding time range (see time range 2 in Figure 6.2). We call the resulting set of core developers *privileged developers* or D_{priv}, for brevity.

As our approach to identify core developers does not require any manual effort (except for determining which event belongs to which category), it can be performed for each GitHub project fully automatically.

6.2.3 Validating Our Set of Core Developers

After identifying the privileged developers, we need to validate this selection. For this purpose, we ask the following research question:

4 We have conducted a sensitivity analysis to explore whether we should consider extended events in addition, as users with triage permission can already make some minor decisions (e.g., decide which issue is labeled as a bug). We discuss this in Section 6.3.3.

> **RQ$_2$:** Is the set of privileged developers D_{priv} a sound approximation for the set of core developers?

To answer this question, we need to obtain a reliable set of core developers for a project. Although there are many state-of-the-art classification methods for identifying core developers (as presented in Section 6.1.1), all these methods might end up in different classifications, making it hard to validate our approach with respect to them. Moreover, as we do not know the accuracy of these methods, we also cannot rely on them for the validation of our approach. Instead, we search for projects that provide publicly available, project-reported maintainer or committer lists, because the developers listed in such lists are (as an official source of information) reliably considered as core developers by the project itself. As projects that collect and provide this information are rare, we can perform this assessment only on a subset of our subject projects. Even if such lists are available, there are different formalisms to publish them. Whereas some projects provide lists with different team-member categories in their repositories and keep them up to date (e.g., Node.js provides a steadily updated list of "technical steering committee", "collaborators", and "triagers"[5]), other projects just provide a static list of contributors on external project websites (e.g., as for project Angular[6]) without any historic resolution. Another issue with the latter is to find out who is really a developer and who is just organizational staff, especially if there is a company behind the OSS project. Due to these inconsistencies, we take manual efforts to determine which parts of which lists are relevant for our study. We do not obtain time-resolved lists, but consider only the most recent time range that we analyze for a project. We call the set of developers that have their mandates publicly documented in project-reported lists *documented developers* or D_{doc}.

Note that neither D_{priv} nor D_{doc} are guaranteed to be complete: We cannot be sure that the project-reported lists are regularly updated and really contain all core developers (e.g., some core developers might not want to be listed there), and we can also not be sure that D_{priv} is complete as there could also be core developers that do not have the respective permissions or do not trigger corresponding events. To investigate whether our automatic procedure of extracting D_{priv} is sound, we cross-check it with D_{doc}. For this purpose, we use the *Jaccard index* as a similarity measure and the measures *completeness* and *soundness* as defined as follows:

$$\text{jaccard}(D_{doc}, D_{priv}) = |D_{doc} \cap D_{priv}| \,/\, |D_{doc} \cup D_{priv}| \tag{6.1}$$

The Jaccard index lies between 0 and 1, higher values indicating higher similarity.

$$\text{complete}(D_{priv} \mid D_{doc}) = |D_{doc} \cap D_{priv}| \,/\, |D_{doc}| \tag{6.2}$$

$\text{complete}(D_{priv} \mid D_{doc})$ is the completeness of D_{priv} with respect to D_{doc}, that is, the proportion of developers in D_{doc} that are also part of D_{priv}.

$$\text{sound}(D_{priv} \mid D_{doc}) = |D_{doc} \cap D_{priv}| \,/\, |D_{priv}| \tag{6.3}$$

$\text{sound}(D_{priv} \mid D_{doc})$ is the soundness of D_{priv} with respect to D_{doc}, that is, the proportion of developers in D_{priv} that are also part of D_{doc}.

5 `https://github.com/nodejs/node/blob/master/README.md#current-project-team-members` (accessed at 2022-03-14)
6 `https://angular.io/about?group=Angular` (accessed at 2022-03-14)

In addition to validating D_{priv} with respect to D_{doc}, we also need to compare D_{priv} to the outcomes of state-of-the-art classification methods, to obtain a deeper understanding of how our approach performs (see Section 6.2.5). However, before we do so, let us first introduce the state-of-the-art classification methods that we investigate.

6.2.4 Developer Classification

Since the set of privileged developers cannot be used to identify potential future maintainers before they get any privileges in the project, and since the necessary data for identifying the set of privileged developers is not necessarily available for projects that use other social coding platforms than GitHub, other classification methods for identifying core developers are still essential. As discussed in Section 6.1.1, many unsupervised classification methods for identifying core developers do exist. For the network-based methods, there are also various network-construction methods. This yields our main research question:

> **RQ$_3$:** Which metrics and network-construction methods are most accurate in classifying developers into core and peripheral?

On the one hand, we investigate established count-based metrics. Particularly, we choose the two most frequently used metrics of this sort (based on our overview of the classification metrics used in the literature in Table 6.2):
Commit count: the number of commits a developer has made in a certain time range.
LOC count: the number of lines of code (LOC) a developer has changed in a time range.
The more central a developer is, that is, the more commits a developer has made or the more LOC the developer has changed, the more likely the developer is a core developer.

On the other hand, to capture the co-coding and co-communication activities of developers, we investigate network-based metrics and different network-construction methods, which we describe in what follows.

Network Types & Network Construction We investigate three different types of developer networks. All three of them have in common that the vertices represent developers and the edges represent relations among them. However, the different network types differ in the type of relations:

Cochange: Two developers are connected by an edge when they have edited the same file within the same time window, as introduced in Section 2.3.1 [GHJ98; JMA+15; JSS11; PBD+14; PD08; ZZW+05]. This network type provides a pure technical view.

Issue: Two developers are connected by an edge when they have contributed to the same issue or pull request (e.g., commenting, reviewing, closing, labeling, etc.) within the same time window, as introduced in Section 2.3.1 [CWL+06; HKC+11; MJT+22; PBD+14; SGR11]. This network type provides a social view.

Cochange+issue: Contains edges from both above mentioned relations. This network type provides a socio-technical view.

Table 6.4: The network-construction methods and network centrality metrics that we use in our study.

Network Type	Network Vertices	Directedness	Simplicity	Time Range	Time Windows	Centrality Metric
Cochange	All contributors	Directed	Simplified	3 months	Sliding	Degree centrality
Issue	Code contributors	Undirected	Unsimplified	6 months	Subsequent	Eigenvector centrality
Cochange+issue				9 months		Hierarchy centrality
				12 months		

As also users of a project (not being developers) can contribute comments to issues, we investigate two different cases (i.e., sets of vertices) when constructing issue networks:

All contributors: Construct networks with all users participating in issues, regardless of whether they contributed to the source code.

Code contributors: Only users are considered who have contributed to the source code (i.e., who have authored a commit either in the current or in any previous time range). All other users are removed from the network.

Note that the differentiation between *all contributors* and *code contributors* affects only the network types *issue* and *cochange+issue*, as *cochange* networks by construction contain only developers who contributed to the source code.

To construct the networks, we split the project data into subsequent time windows of the same length, as common in the literature. We chose to investigate time windows of 3 months [BGD+07; JAH+17; MJT+22; TPS+19], 6 months [HKC+11; MW11; PBD+14], 9 months, and 12 months [RG06]. Shorter time windows would be threatened by short-term fluctuations (e.g., developers being ill or on vacation). Using larger time windows would neglect project dynamics and developer turnover [FSV20]. As edges that cross time-window boundaries are neglected, we additionally investigate sliding windows, that is, instead of using subsequent time windows only, we shift the subsequent window by half the time window, such that subsequent windows overlap and cover the edges that are neglected when not using such a sliding-window approach [JAM17], as described in Section 2.3.2.

When constructing networks, we vary between the network-construction parameters *directedness* and *simplicity* (see Section 2.3.1): In a directed network, we consider the order of interactions (e.g., who replies to whom, including transitive relationships as separate edges), whereas in an undirected network, we ignore the temporal relationship. In a simplified network, only one edge per direction is allowed between one pair of developers (no loops), whereas in an unsimplified network, multiple edges between one pair of developers and loops are allowed. In Table 6.4, we provide an overview of all possible choices during network construction as well as the network-based classification metrics that we use, which we explain in the following.

Network-Based Classification To classify developers into core and peripheral, we use three established network metrics, which are widely used in the literature (as can be seen in Table 6.2) and which cover different aspects of centrality. In particular, we selected the following network centrality metrics, which we have introduced in Section 2.3.3:

- *degree centrality*, a local centrality metric,
- *eigenvector centrality*, a global centrality metric that considers the importance of the developers a developer is interacting with, and
- *hierarchy centrality*, a metric that considers the community structure of a network.

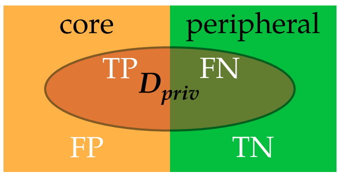


Figure 6.3: Visualization of *true positives* (*TP*), *false positives* (*FP*), *true negatives* (*TN*), and *false negatives* (*FN*). The ellipsis represents the set of privileged developers D_{priv}, the two rectangles represent the core/peripheral classification of the used classification method (i.e., the sets of developers that are classified as core or peripheral, respectively).

For background information on these three centrality metrics, we refer to Section 2.3.3. All the three network centrality metrics we use have already been used in previous work on developer classification and received a high agreement in the perception of surveyed developers [JAH+17].

On each of the metrics, for each time range, we apply the 80% threshold, which is widely used (see Section 6.1.1). That is, developers whose centrality value is in the upper 20% quantile are considered as core, the remaining ones as peripheral. For data processing, network construction, and centrality computation, we use the library CORONET, which we have introduced in Section 2.4.

6.2.5 Assessing the Classification Accuracy

To assess the accuracy of the classification methods, we use the set of privileged developers D_{priv} as a point of reference. There are multiple reasons behind this choice. On the one hand, there are practical reasons: As identifying privileged developers is not applicable on projects that use other social coding platforms than GitHub, and since the set of privileged developers cannot be used to identify potential future core developers before they get any privileges, the state-of-the-art classification methods are still necessary. In such cases, it would be helpful to know for practitioners and researchers which of the state-of-the-art methods are closest to our approach on identifying privileged developers. On the other hand, D_{priv} is constructed based on the privileges developers made use of in issue events. Hence, these developers must have received the corresponding permissions in the project from the project's maintainers, which incorporates trustworthy information that comes from a project itself, whereas the established state-of-the-art classification methods do not consider such information.

For assessing the classification accuracy, we compute precision, recall, and F1 score, based on the following notions (see also Figure 6.3):

***True positives* (*TP*):** developers that are classified as core and are part of D_{priv}.

***False positives* (*FP*):** developers that are classified as core but are not part of D_{priv}.

***True negatives* (*TN*):** developers that are classified as peripheral and are not part of D_{priv}.

***False negatives* (*FN*):** developers that are classified as peripheral but are part of D_{priv}. In addition, developers that are part of D_{priv} but that are neither classified as core nor peripheral are also considered to be a false negative (e.g., developers who did never commit but are part of D_{priv} due to triggering privileged issue events are missing in commit-data-based classifications).

Note that the definitions of *FN* and *FP* are specific to our setting: *FN* and *FP* are not defined with respect to D_{priv}, but with respect to the classification. "negative" corresponds to "peripheral"; "positive" corresponds to "core". With that, we compute the following evaluation measures:

$$\text{precision} = TP \,/\, (TP + FP) \tag{6.4}$$

$$\text{recall} = TP \,/\, (TP + FN) \tag{6.5}$$

$$\text{F1} = 2 \cdot (\text{precision} \cdot \text{recall}) \,/\, (\text{precision} + \text{recall}) \tag{6.6}$$

Also note that the choice of the point of reference for the comparison of D_{priv} and the classification results is arbitrary in our case: If we would take the results of the state-of-the-art classification methods as point of reference for assessing the accuracy of D_{priv}, the values of precision and recall would only be swapped (since *FP* would become *FN* then, and vice versa), and, consequently, the F1 score would stay the same. Nevertheless, for the reasons stated above, we chose D_{priv} to be the point of reference in our comparison, to assess the accuracy of the state-of-the-art classification methods.

6.3 Validation Study

In this section, we present the results of our empirical study. We start with an overview of our subject projects, provide the results of our validation of D_{priv}, and report on the assessment of the accuracy of the unsupervised developer-classification methods.

6.3.1 Subject Projects

We investigate 25 highly active software projects hosted on GitHub, covering various project sizes, domains, and numbers of contributors and participants in issues, as the descriptive statistics in Table 6.5 indicate. We primarily selected very popular projects that are among the most starred GitHub projects in 2020.[7] Most of these projects have also been considered in previous work on GitHub-related research topics. In order not to only analyze popular projects that have a high developer activity, but also cover projects that have a comparably small number of developers, we added project DTP.

6.3.2 Time Difference Between Privileged Events

In RQ_1, we ask how long the typical time difference is between a developer's privileged events. In most of the projects, the median time difference is between 0 and 4 days; even the upper quartile of the time differences is below 10 days (except for jQuery, for which it is about 3 weeks). Although there are extreme outliers, for which there are up to 2 000 days between the privileged events of a developer, in most cases, developers make use of privileged events, at least, multiple times a month. As we can see in Figure 6.4, for more than 88% of the privileged developers the median time difference between privileged events is smaller than

7 https://www.attosol.com/top-50-projects-on-github-2020/ (accessed at 2020-09-09)

Table 6.5: Descriptive statistics of our subject projects.

Subject Project	Investigated Time Period	# Commit Authors	# Issue Participants	# Commits	# Issues incl. PRs	Project Domain	Programming Languages
ANGULAR	2014-09–2020-09	667	22 859	12 349	38 502	Web dev. platform	TypeScript
ATOM	2012-01–2020-12	298	21 047	15 627	21 138	Text editor	JavaScript
BOOTSTRAP	2011-08–2020-12	219	24 744	2 266	31 735	Web frontend framew.	JavaScript, HTML
DENO	2018-05–2020-12	348	3 070	3 417	8 760	Runtime for JavaScript	Rust, JavaScript, TypeScript
DTP	2018-01–2020-04	16	73	633	859	Framew. f. data transf.	Java
ELECTRON	2013-05–2020-12	392	15 559	10 664	26 733	App. dev. framew.	C++, TypeScript
FLUTTER	2015-03–2020-12	683	34 460	13 367	72 504	UI dev. kit	Dart
jQUERY	2010-09–2020-12	244	3 118	2 675	4 723	JavaScript library	JavaScript
KERAS	2015-03–2019-11	716	12 688	3 471	13 468	Deep learning API	Python
KUBERNETES	2014-06–2020-12	2 408	23 220	38 619	97 218	Container managem.	Go
MOBY	2013-01–2020-12	1 154	29 083	14 072	41 731	Softw. containerization	Go
NEXTCLOUD	2016-06–2020-09	355	9 510	9 718	22 689	Cloud server	PHP, JavaScript
NEXT.JS	2016-10–2020-12	867	11 087	3 891	15 344	React framew.	JavaScript, TypeScript
NODE.JS	2014-11–2020-02	1 793	13 190	12 118	31 372	JavaScript runtime env.	JavaScript, C++, Python
OPENSSL	2013-05–2019-12	400	3 303	8 722	10 639	Crypto library	C, Perl
OWNCLOUD	2012-08–2019-10	393	10 141	18 274	36 178	Cloud server	PHP, JavaScript
REACT	2013-05–2020-12	796	16 056	6 921	20 252	JavaScript library	JavaScript
REDUX	2015-06–2020-12	228	4 123	701	3 931	Container f. JavaScript	TypeScript, JavaScript
REVEAL.JS	2011-06–2020-10	141	2 861	1 090	2 762	HTML present. framew.	JavaScript, HTML
TENSORFLOW	2015-11–2020-12	1 519	35 781	55 499	45 652	Machine learn. framew.	C++, Python
THREE.JS	2010-04–2020-12	954	8 280	15 999	20 845	JavaScript library	JavaScript, HTML
TYPESCRIPT	2014-07–2020-12	467	18 397	17 934	40 973	JavaScript language	TypeScript
VS CODE	2015-11–2020-12	1 001	67 882	49 814	111 073	Integrated dev. env.	TypeScript
VUE	2016-04–2020-11	217	8 754	2 256	9 325	JavaScript UI framew.	JavaScript
WEBPACK	2012-05–2020-12	501	13 091	5 671	11 710	Bundler for modules	JavaScript

6 months. This holds for all projects except for VUE, for which this time difference is only for 75% of the developers smaller than 6 months, but also reaches 88% at about 7.5 months.

Answer to RQ$_1$: The majority of privileged developers trigger privileged events, at least, once within a couple of weeks. More than 75% of these developers have a median time difference less than 3 months between their privileged events.

As a consequence, determining the set of privileged developers D_{priv} based on privileged events with time windows of 3–12 months is justified. That is, due to applying our temporal smoothness assumption (i.e., also looking at the previous and subsequent range, see Figure 6.2), our approach is robust to time differences of up to three times the window size, ending up in covering core developers who use their permission only once in 9–36 months. As for all but one project more than 88% of the developers using privileged events have a median time difference of less than 6 months between these events, we focus on presenting the results using 6-months ranges. For other time ranges, we refer to our supplementary website.[8]

8 https://se-sic.github.io/paper-developer-classifications/

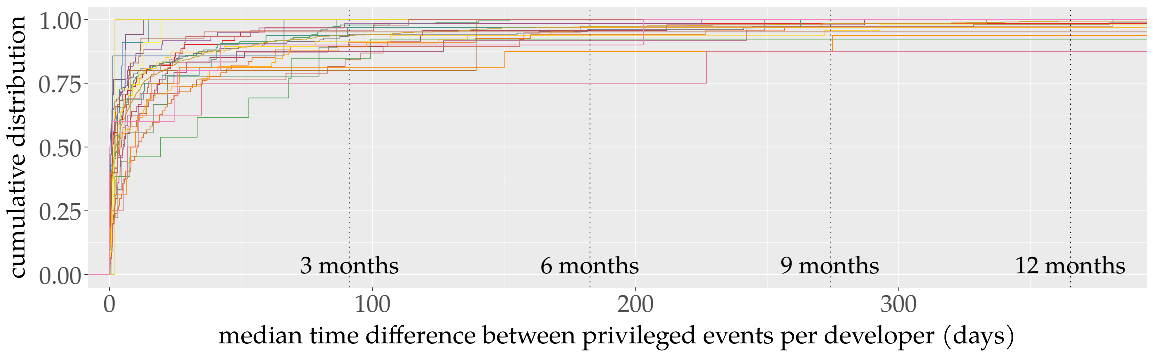


Figure 6.4: Cumulative distribution of the median time difference in days between privileged events of a single developer. Each line represents the developers of one subject project.

6.3.3 Validity of the Set of Privileged Developers D_{priv}

To answer RQ_2, we check for the validity of the set of privileged developers D_{priv}. For this purpose, we compare D_{priv} in the most recent time range with project-reported lists (D_{doc}). We were able to obtain D_{doc} for 12 out of our 25 subject projects.

In Table 6.6, we report the sizes of D_{doc} and D_{priv} for the last-analyzed 6-months time range of each project. When considering only privileged events to detect core developers, we can see that the size of D_{priv} is, in most projects, smaller or nearly equal to the size of the project-reported lists. However, when considering privileged and extended events (see D_{priv+} in Table 6.6) to detect core developers, in most projects, we are able to extract a much higher number of core developers than reported by the projects' lists. This reinforces our decision that core developers are the developers that have the permission to trigger privileged events. Nevertheless, the similarity of D_{priv} and D_{doc} is highly project-dependent, as the Jaccard Indices between 0.03 and 1.0 indicate. There are two outliers, though: Whereas KUBERNETES has way more developers in D_{doc} than in D_{priv}, the list reported by NEXT.JS only contains very few developers. When looking at completeness, we can see that we are able to gather up to 67% of the project-reported developers in D_{priv} (for REVEAL.JS, where only one developer is reported, we reach even 100%). Soundness, on the other hand, reaches up to 94% (for REVEAL.JS even 100%), showing that the vast majority of developers in D_{priv} are also in D_{doc}. We obtain similar results when extracting D_{priv} using other time-window lengths; for more details, we refer to our supplementary website.[8]

Answer to RQ_2: The completeness of D_{priv} with respect to D_{doc} seems to be rather small. This is to be expected as we cannot be sure that the project-reported lists are up to date. More importantly, the vast majority of developers in D_{priv} is documented in D_{doc}, which shows that our procedure of extracting D_{priv} is sound.

6.3.4 Classification-Method Accuracy

Finally, to answer RQ_3 about which unsupervised classification methods are most accurate in automatically classifying developers into core and peripheral, we assess their accuracy using

Table 6.6: Validation of the procedure to extract privileged developers for the 12 projects for which we found project-reported lists. (We collected the lists only once per project, temporally close-by to the end of the latest 6-months time range that we analyzed.)

	Angular	Electron	Keras	Kubernetes	Moby	Next.js	Node.js	OpenSSL	React	reveal.js	Vue	webpack
$\lvert D_{doc} \rvert$	30	17	5	238	21	6	108	18	9	1	3	4
$\lvert D_{priv} \rvert$	20	12	5	51	9	25	34	11	10	1	3	6
$\lvert D_{priv+} \rvert$	40	22	10	1125	9	27	63	199	392	2	3	7
jaccard(D_{doc}, D_{priv})	0.43	0.61	0.43	0.08	0.36	0.03	0.29	0.53	0.46	1.00	0.20	0.25
complete$(D_{priv} \mid D_{doc})$	0.50	0.65	0.60	0.09	0.38	0.17	0.30	0.56	0.67	1.00	0.33	0.50
sound$(D_{priv} \mid D_{doc})$	0.75	0.92	0.60	0.43	0.89	0.04	0.94	0.91	0.60	1.00	0.33	0.33
jaccard(D_{doc}, D_{priv+})	0.49	0.63	0.25	0.08	0.36	0.03	0.50	0.09	0.02	0.50	0.20	0.22
complete$(D_{priv+} \mid D_{doc})$	0.77	0.88	0.60	0.44	0.38	0.17	0.53	0.94	0.78	1.00	0.33	0.50
sound$(D_{priv+} \mid D_{doc})$	0.58	0.68	0.30	0.09	0.89	0.04	0.90	0.09	0.02	0.50	0.33	0.29

D_{doc}: project-reported list,
D_{priv}: core developers based on privileged events,
D_{priv+}: core developers based on privileged+extended events

precision, recall, and F1. It turned out that there is only a small-to-zero difference between using time windows of 3, 6, 9, or 12 months or using sliding or subsequent windows. Also the difference of the results when using directed or undirected or simplified or unsimplified networks is marginal. For this reason, we present here only the results for subsequent 6-month windows, using unsimplified and directed networks, and briefly put each of the different parts of these results into context; all other results are available on our supplementary website.

In Figure 6.5, we provide an overview of the accuracy of the different classification methods across all subsequent time ranges of all 25 subject projects. The precision is higher for count-based classifications (median precision between 0.6 and 0.8) and cochange-network-based classifications (median precision around 0.5) than for issue-based (median precision around 0.04) or cochange+issue-based classifications (median precision around 0.04). That is, classifications based on commit data (as in the count-based approaches and on the cochange network) contain a higher percentage of correctly classified core developers over all classified core developers, as compared to the classifications using networks derived from issue data. This may be due to core developers extensively contributing to the source code, whereas in issue networks there are also highly active commenting developers, who appear central in the issue network and are, therefore, wrongly classified as core. In contrast, the recall is higher for classifications based on issue (and cochange+issue) networks (median recall around 1.0) than for commit-data-based classifications (median recall between 0.3 and 0.55). That is, the proportion of developers from D_{priv} that are contained in the classified set of core developers is high, sometimes even close to 100%. This is not unexpected as D_{priv} was derived from issue data. More interestingly, these results indicate that the developers who trigger privileged events also are central in issue networks and may also extensively participate in issues. In general, the number of developers classified as core is higher on issue networks

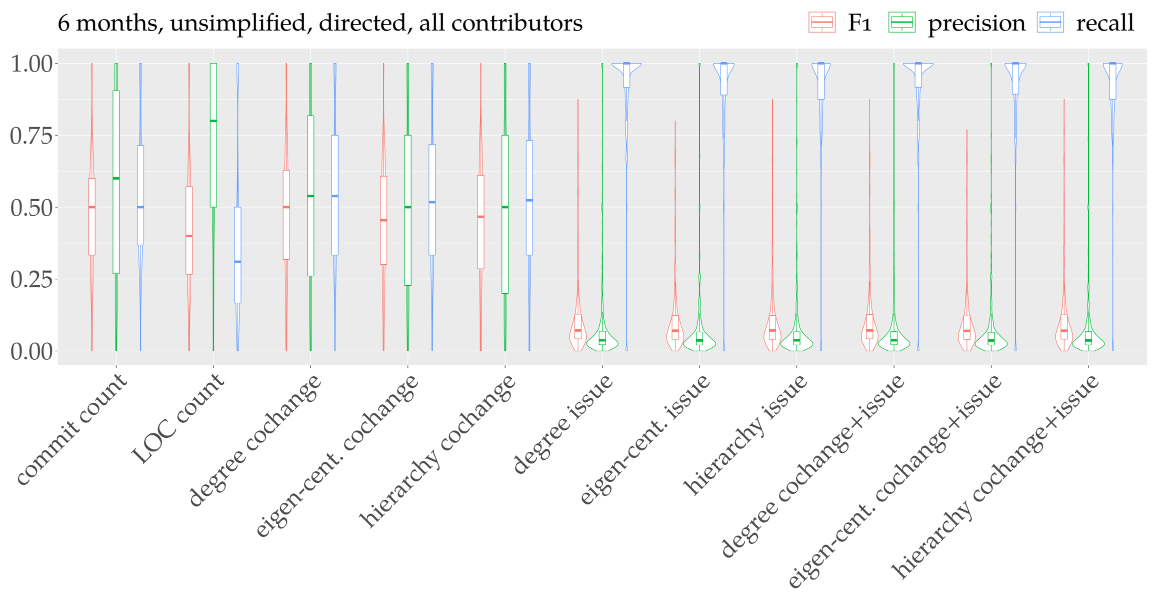


Figure 6.5: F1, precision, and recall for each classification method with respect to D_{priv}.

than for commit-data-based classifications, and also higher than in D_{priv}. This coincides with the overall number of developers, which is in issue data way higher than in commit data, which may be a reason for the low precision of the approaches that use issue data.

It is also worth noting that the recall for methods that rely on commit data is lower than for the remaining methods, as there are developers in D_{priv} that are not part of the commit data and, therefore, neither classified as core nor as peripheral. To account for that, we investigated how many of the developers in D_{priv} are not classified, at all. Across all projects, a median of about 17% of the developers of D_{priv} are not classified by commit-data-based methods. This mainly happens because of two reasons: (1) Core developers may focus on maintenance tasks (such as reviewing and merging) and do not contribute to the source code at all. (2) Some developers only contribute to documentation files (such as README.md), which are not covered in our commit data, as we only keep track of files that contain source code.

To compare the overall performance of the different classification methods, we look at the F1 score. The commit-data-based classification methods have comparably high median F1 scores between 0.4 and 0.5, whereas all the remaining classification methods have much lower median F1 scores of about 0.07. There is almost no difference in the accuracy between issue networks (median F1 around 0.07) and cochange+issue networks (median F1 around 0.07). The reason is that the issue network dominates the cochange+issue network, as there are way more issue discussions than commits (which can also be seen in Table 6.5). Also the difference in the accuracy of degree centrality, eigenvector centrality, or hierarchy centrality is marginal, which is moreover independent of the network type (mostly degree centrality performs best, in some projects hierarchy centrality performs best). So, in what follows, we neglect the different network metrics and also the cochange+issue networks to focus on comparing count-based, cochange-based, and issue-based classification methods.

To obtain an overview of the overall performance on different projects, we ranked the F1 score (using the median to aggregate across time ranges) of each classification method for each

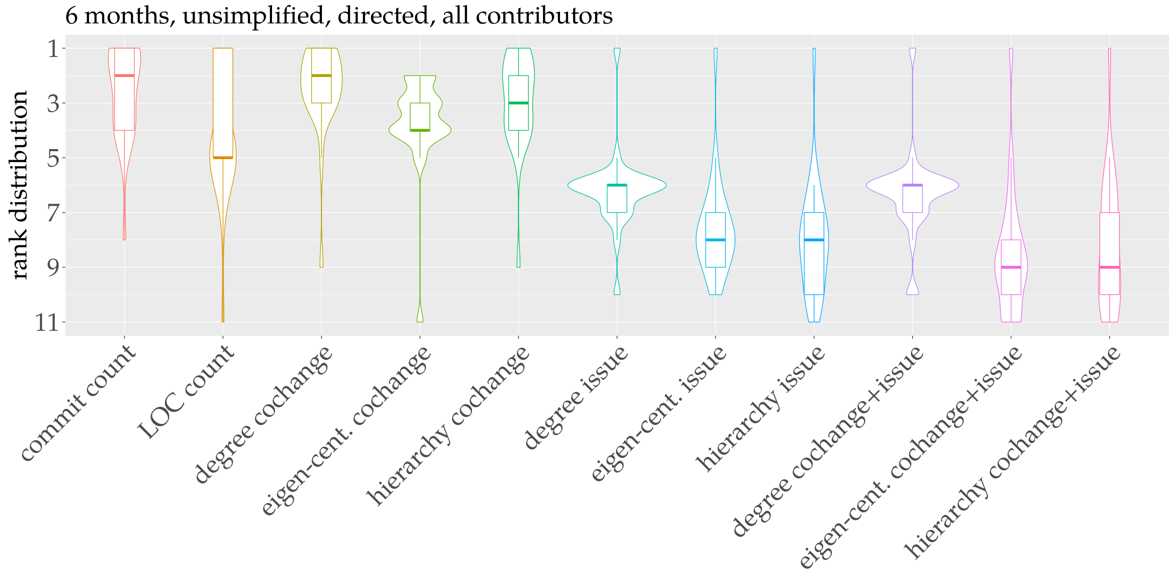


Figure 6.6: Distribution of the rank of the different classification methods. That is, for each project, we ranked the median F1 scores of all classification methods. The classification method(s) that had the highest median F1 on the project receive rank 1, the method(s) that had the lowest median F1 in the project receive rank 11.

project. The classification methods that resulted in the highest aggregated F1 score on a project received rank 1, the ones that resulted in the lowest aggregated F1 score on a project received rank 11 (as we investigate 11 different classification methods). In Figure 6.6, we show the distribution of the ranks of the classification methods across all projects. In most projects, the commit-count-based method or the cochange-based methods perform best. However, there are also projects, in which the issue-based classifications yield higher F1 scores (e.g., projects Flutter or DTP; we illustrate accuracies for each project on our supplementary website). These projects have a comparably low number of commits as opposed to a comparably high number of issues.

The overall picture slightly changes when restricting the issue data to code contributors (see Section 6.2.4). In this case, the issue-based classifications end up with a lower recall (median values around 0.8) but a higher precision (median values around 0.3) and also a higher F1 score (median values around 0.4) than when considering all contributors (see median values stated above). So, when considering only code contributors, the issue-based methods are almost as accurate as the commit-data-based methods, sometimes even more accurate.

Answer to RQ$_3$: In summary, using commit data or cochange networks for classifying core developers performs best. Though, which of these classification methods performs best, is project-dependent. Issue networks often are distorted by users who intensively participate in discussions. Different network-construction methods or time-window lengths do not make a substantial difference for classifying developers into core and peripheral.

6.4 Discussion

As our results for RQ_1 indicate, privileged developers make use of their privileges very often, mostly multiple times within a couple of weeks. Therefore, extracting privileged events within time ranges of several months is sufficient to identify the set of privileged developers. Our results for RQ_2 demonstrate that the set of privileged developers is a sound approximation for the set of core developers, which might not be complete, though, as there can be core developers that do not make use of their privileges or that did not obtain the respective privileges. This is the point at which our approach is stretched to its limits: We cannot identify core developers before they receive privileges. Here, state-of-the-art classifications methods become necessary again, which is why we evaluated their accuracy with respect to our approach, to provide researchers and practitioners the option for selecting an appropriate method for their specific use case. Our results for RQ_3 indicate that *unsupervised* classification methods that use commit data perform slightly better than methods that use issue data. This demonstrates that issue discussions are dominated by users who ask questions or use issues to retrieve knowledge about the project. Using the commit count, which is known to provide only a limited view on organizational structure [JAH+17], seems to perform similarly accurate for classifying core developers than various network-based classification methods. This indicates that developers' work on the source code (i.e., having a high coding activity) is, at least, as relevant for becoming a core developer as their interaction with other people. We arrive at a similar conclusion when looking at the used network-construction methods and classification metrics, since they do not make a substantial difference. Sometimes, a more simple, local centrality metric (such as degree centrality) performs even better than eigenvector centrality as a global centrality metric. Nevertheless, these classification metrics perform all pretty similarly, and it is also project dependent as well as data-source dependent which one performs most accurate with respect to the set of privileged developers. The fact that the different methods perform similarly well with respect to our set of privileged developers also indicates that our approach of identifying privileged developers, indeed, is a reasonable approximation for the set of core developers. All in all, instead of recommending a specific method, our main goal is to inform researchers and practitioners about the accuracy of the state-of-the-art unsupervised classification methods.

Our approach provides a viable basis for future research. On the one hand, since receiving a higher level of permissions in a project is also a sign of trust within the community, using the set of privileged developers, we can obtain more information about the characteristics that a developer needs to become a core developer. In particular, we can identify in which time range developers have received their privileges (or, at least, when they have used it for the first time) and search for relevant characteristics of these privileged developers (also considering the role within the network and community structure) before becoming a privileged developer. Such information can be used to improve the search for future candidates to take on more responsibility in the project. In addition, we can also check how the relevant characteristics and the position in the network change after a developer has become part of the set of privileged developers, allowing future researchers to investigate how the activities of a core developer alter after moving up the ladder. This allows for improving community engagement, identification of future project maintainers, and project stability and evolution.

On the other hand, our automatic approach to identify core developers could be used for the development of *supervised* learning methods for identifying core developers. Up until today, to the best of our knowledge, the lack of sufficient volumes of ground-truth data has prevented *supervised* learning methods from being developed. Most of the developer-classification methods used in the literature are either unsupervised or have manually configured thresholds. Hence, project data is only used for evaluation but not for model fitting. Yet, our automatic approach to identify core developers based on privileged issue events could be used for supervised learning methods. To demonstrate the possible benefit of our approach with respect to this research direction, we provide a perspective on supervised classification in the following section by means of a small explorative experiment.

6.5 A Perspective on Supervised Classification

As already explained in Section 6.4, state-of-the-art unsupervised developer-classification methods mostly rely on manually configured thresholds or other manual steps. Although we devise an automatic approach to identify core developers based on privileged issue events, it is not capable of identifying potential future core developers before they get any privileges, and it is also not suitable for projects that do not use social coding platforms such as GitHub. Hence, there is still a need for supervised classification methods. Therefore, using our automatic approach to identify core developers based on privileged issue events opens the door for future research to obtain a higher level of maturity when identifying core developers by automated means: Whereas manually configuring the optimal classification thresholds would be a tedious task, supervised learning can be used to automatically learn classification thresholds. Also, combining different classification methods (that use different data and classification metrics) by hand would be impracticable. Instead, learning the weights for combining multiple methods and automatically detecting which combinations are beneficial and which ones not would be possible in a holistic supervised-learning approach. To sketch out the possible benefit of using our automatic core-developer identification approach as a *foundation* for future research in this direction, we conducted a small explorative experiment on how to learn the classification threshold in a binary classification task for a single project or for a set of projects, which we describe and report on in what follows.

Experiment Setting As a preliminary exploration of the potential that supervised learning offers, we pose the problem of role identification as a binary classification task. Instead of using manually configured thresholds, our idea is to learn a function $f\colon d \to \{0, 1\}$ that maps a developer d to the core or peripheral class. While some of the classification methods that we used in our validation study are certainly better than others, it is plausible that each of them captures a different dimension of developer roles. For this reason, we construct the input to the classification model to be a combination of all data and metrics used in our validation study. We experimented with several classification models but found that the random-forest classifier [CG16] with a maximum depth of five has the best generalization performance. Due to highly imbalanced classes, we augment the minority class in the training dataset by created synthetic data points using the SMOTE technique [CBH+02]. To evaluate the generalization performance of the model, we apply standard cross validation and split the data into separate training and testing sets. We report results in terms of F1 score on the test set for the random-forest classifier.

Prediction Scenarios We explore two different prediction scenarios. In the first scenario, we apply k-fold cross validation by splitting the developers randomly into ten groups. We then train ten different models such that each model is trained on nine groups and tested on the remaining tenth group. The prediction performance is then averaged over the ten models. This scenario tests how well the model can generalize to a new developer when the model is able to learn from training examples of the roles from every project. For this purpose, we made sure that the test set contains only developers that have not been part of the training set. In the second scenario, we split the developers according to projects so that the training set consists of developers from all projects except the one which forms the test set. In total, 25 models are trained, where each project appears once in the test set and the prediction metrics are averaged. This scenario is more difficult because it tests the model's ability to learn general knowledge about roles in other projects.

Experiment Results For the first scenario, we get an F1 score of 0.73 for our combined model, whereas for the unsupervised methods based on a single metric we get lower F1 scores (0.3–0.5). Similarly, for our second scenario, our combined model (0.62) clearly outperforms each of the unsupervised methods (0.2–0.4). Thus, our explorative small experiment indicates that combining various data sources and socio-technical metrics can help learning promising classification models.

Summary As our explorative experiment showed, our approach holds the potential to become a foundation for future work on training supervised classification models for the identification of core developers. Consequently, based on the outcomes of this experiment, we formulate two hypotheses that should serve as a starting point for future research on this topic, to sum up our perspective on supervised classification:

Hypothesis 1: Developer-classification models that leverage supervised learning are significantly more accurate than state-of-the-art developer-classification methods.

Hypothesis 2: Automatically combining data from different data sources (e.g., commits and issues) and different classification metrics (e.g., the metrics described in Section 6.2.4) as well as learning appropriate weights for combining multiple methods outperforms state-of-the-art developer-classification methods.

6.6 Threats to Validity

The validity of the results of our validation study may be threatened in various directions (as always in empirical studies). We have grouped the potential threats into four categories that were introduced by Cook and Campbell [CC79], as suggested by Wohlin et al. [WRH+12].

Construct Validity Different GitHub projects assign permission levels differently: Some are more liberal, others are more strict. Also individual users use their permissions to a different extent. Thus, relying on the use of privileged events could threaten the validity of computing the set of privileged developers D_{priv}. To mitigate this threat, we investigated two different permission groups (privileged vs. privileged+extended) and four different lengths for the time ranges.

Another threat concerns the way we construct issue networks, as users who do not contribute to the source code may still be part of the issue data. We alleviate this by investigating two cases: When considering only code contributors, the issue-based classification methods perform better than on all contributors, but the overall picture only varies slightly.

Moreover, the network-construction procedure per se may affect our results. To account for that, we investigated different network-construction methods (i.e., directedness, simplicity, time ranges, and time windows, as summarized in Table 6.4). Our results confirm that the choice of the network-construction method does not make a substantial difference.

Conclusion Validity Based on the outcomes related to RQ_2, we conclude that D_{priv} is a sound approximation for the set of core developers (that might not be complete, though). This conclusion relies on the reliability of D_{doc} (i.e., project-reported lists). As these project-reported lists are maintained by the project itself, they are an official source of information declaring who is acting as a core developer in a project. Such project-reported lists may be out of date or incomplete (as already discussed in Section 6.2.3), though. This is why we put on manual efforts to search for these lists and check their relevance for our study. In most cases, when these lists are maintained even within the repository, the commit history shows that they get regularly updated. Still, these lists do not directly map to the time ranges that we used for validating D_{priv}. Since this mapping incorporates substantial manual effort, we performed this validation step only for the last analyzed time range of a subject project. While this might decrease conclusion validity, our manual checks as well as the choice for analyzing the most recent time range ensure that our conclusions are valid, at least, for the time ranges that are close to the last analyzed time ranges, as the project workflows were mostly stable.

Whereas our conclusion that D_{priv} is a sound approximation for the set of core developers is mainly derived from the comparison with D_{doc} (which is an official source of information), the comparison of D_{priv} with the classification outcomes of the state-of-the-art methods (which do not build on officially stated information) provides corroborating evidence.

Internal Validity We do not track non-source-code files such as documentation files in our commit data. This is intended, since we aim at identifying core developers who work on the source code. Yet, non-source-code files usually amount only to a small fraction of the files in a software repository.

When we identify developers, we rely on their names and e-mail addresses, to assign them commits and issue events. This could be problematic if developers use different spelling variants for their names or different e-mail addresses (e.g., when they have configured the name or e-mail address in their GitHub account other than in their Git commit configuration). To address this threat, we use the disambiguation heuristic of Oliva et al. [OSO+12], which has turned out to be reliable in empirical studies [WSS+16], to consider instances that use the same name or the same e-mail address being the same developer, in combination with manual sanity checks. If GitHub users keep their names and e-mail addresses private, we rely on their usernames and on the data that is stored in the commits that are associated with the usernames.

In a similar vein, the detection of bots is a non-trivial task. Therefore, we combine two approaches: We first use the tool BoDeGHa [GDL+21] to automatically detect bots based on their commenting behavior in issues, and then we perform manual adjustments in which we also investigate "bot" substrings in usernames and bot marks visible on GitHub. Nevertheless, even if there are bots that we did not detect, this does not threaten our results significantly. The main incentive for detecting and removing bots is to get rid of the bots that are very active and, therefore, distort the network structure or activity counts. If some bots remain in our data that are as active as a usual developer, there is only a low probability that they would distort our results significantly, since we can handle such bots as we handle real developers.

External Validity As there are various project shapes, one cannot generalize our results arbitrarily to all OSS projects, at large. Though, we analyzed the complete history of 25 subject projects (starting at the earliest point in time for which all the data sources for a project were available). That is, summing up the investigated time periods of all our 25 projects, we analyzed the data of a combined history of about 160 months, which is a substantial amount of data. Due to the high number of network-construction methods that we investigated (since there are 192 possible combinations of the choices in Table 6.4), our comprehensive validation study was computationally expensive, which was the reason for limiting the number of subject projects to 25. Albeit, we analyzed 25 projects of different sizes, project domains, and programming languages, which provides already detailed insights.

The results of RQ_2 are potentially threatened by the fact that we were able to obtain the set of documented developers D_{doc} only for 12 out of our 25 subject projects. That is, we validated the set of privileged developers D_{priv} only for these 12 projects with respect to a project-reported list of core developers. We deliberately decided to keep the remaining 13 projects in our study, though, because, this way, we explicitly account for the fact that there are also projects that do not provide such project-reported lists and that, for such projects, it would be beneficial to automatically identify core developers. Nevertheless, using the 12 projects for which we obtained D_{doc} already shows for a multitude of different projects that our approach of identifying core developers based on privileged events is sound and promising. Note that, for answering RQ_1 and RQ_3, and also for conducting the experiment on supervised learning methods that we have presented in Section 6.5, we used all our 25 subject projects to base the corresponding results on a broader dataset. We did not detect any significant differences in the results between the projects for which we were able to obtain D_{doc} and those for which we did not, which also alleviates this threat.

Our method to construct the set of privileged developers D_{priv} is not directly applicable to other platforms than GitHub. Albeit, other platforms might use similar permissions, so the underlying idea of our method should also be transferable.

6.7 Conclusion

Identifying core developers in OSS projects is beneficial in various occasions: (1) in large-scale software projects, to improve developer coordination and software quality by revealing who makes decisions and who has consolidated project knowledge; (2) for newcomers and peripheral developers, to get in touch with maintainers and core developers; (3) for companies, to decide whether and how to invest in a project or how to efficiently contribute to an OSS project; (4) and for researchers, to obtain deeper insights into organizational structures of OSS projects and their evolution. Identified core developers that are unknown yet to the project leaders may be candidates to take on more responsibility in the project.

As explicit information on who is core developer is rarely available, we devise an *automatic* method for identifying core developers in GitHub projects based on privileged issue events as well as an assessment of the accuracy of state-of-the-art unsupervised classification methods. Even more than recommending a specific method, we aim at informing researchers and practitioners about the performance of the available methods. Our empirical study on 25 GitHub projects reveals that the choice of data source (commit data vs. issue data) matters more than the actual classification metric (e.g., the centrality metric), which is a non-obvious result. Our results shall guide practitioners and researchers to choose an appropriate *unsupervised* classification method and provide a solid foundation for future *supervised* learning methods. For this purpose, we have formulated two hypotheses that should serve as guidance for future work on developer-classification methods.

7 Investigating Organizational Events in Open-Source Software Projects

In open-source software (OSS) projects, the success and sustainability of a project may not only depend on the quality of source code and the corresponding maintenance work, but may also be influenced by external factors that are not directly under the control of the developers. Such external factors could be different kinds of organizational events that happen during the evolution of a software project. For example, this can be forks of a project that emerge during the project's evolution, which can attract users and developers more than the original project, for various reasons, and leads to a competitive situation between the different project forks [NM11; RG12; ZVK20]. A prominent example of such a situation are the projects ownCloud and Nextcloud, where Nextcloud has emerged as a fork of ownCloud in 2016.[1] Also core developers who leave the project can potentially cause serious trouble, for instance, when they take important project knowledge with them, which hinders further advancements of the project and prevents future core developers from keeping the project in business [FSV20; IRO+09; MTM+19; Moc10]. For example, the lead maintainer of project Gogs disappeared for more than a month, which temporally hindered the project development and eventually led to the creation of its fork Gitea.[2] To provide a third example, companies that take over an OSS project or cease their support for a project, such as IBM's withdrawal from OpenOffice in 2014,[3] can also affect the success and sustainability of a project. All these different kinds of organizational events have in common that they can cause changes in the developer participation and, thus, in the organizational structure of the project. Leaving developers and missing project knowledge can, for example, lead to the introduction of bugs or to unmaintained projects [NR17; RGM05]. In the worst case, these organizational events can even lead to project abandonment or failure [CV17]. Therefore, investigating such organizational events, their causes, and their implications can be beneficial for understanding and improving OSS development, and, thus, can be seen as a first step toward avoiding project failure eventually.

In this chapter, we aim at obtaining an understanding of the influence of organizational events on the organizational structure of OSS projects. Understanding how such events influence the organizational structure could further be used to develop mechanisms to avoid critical consequences of events with negative connotations (such as the introduction of bugs or, in the worst case, project failure and project abandonment). In particular, we leverage

1 Frank Karlitschek: "Nextcloud", 2016: https://karlitschek.de/2016/06/nextcloud/ (accessed at 2023-11-08)
2 GitHub issue: "Is Gogs dead?", 2015: https://github.com/gogs/gogs/issues/1304/ (accessed at 2023-11-08)
3 Bruce Byfield: "LibreOffice, OpenOffice, and rumors of unification", 2014: https://www.linux-magazine.com/Online/Blogs/Off-the-Beat-Bruce-Byfield-s-Blog/LibreOffice-OpenOffice-and-rumors-of-unification/ (accessed at 2023-11-30)

the network representation of developers' collaboration and communication activities to obtain insights into how different kinds of organizational events influence the organizational structure of a project. By means of an exploratory empirical study on a diverse set of case studies [RHR+12], we investigate exemplary cases of organizational events and analyze whether and how these events are reflected in various developer-network characteristics. In the long run, developer network-characteristics could be used to reveal particular specifics (e.g., a reduced information flow and a lack of robustness to developer turnover in consequence of a sudden loss of scale-freeness) that potentially threaten the organizational structure of the project and, thus, the success and sustainability of the project. Knowing about such specifics could be used to develop an early-warning mechanism or even to develop counteractions against critical organizational events, to avoid negative consequences of such events.

All in all, we make the following contributions:

- An exploratory investigation of different kinds of organizational events on an exemplary basis for 8 case studies (i.e., 8 OSS projects and some of their forks), analyzing whether organizational events are reflected in various characteristics of developer networks.
- Insights into the evolution of developer-network characteristics, which could be useful for answering various research questions on the organizational structure of OSS projects.
- Hypotheses regarding which network characteristics might be promising indicators for detecting specific organizational events that affect the structure of developer networks.
- A foundation for future work on developing measures to detect organizational events and to develop early-warning mechanisms and countermeasures to avoid potentially negative consequences of organizational events.
- A replication package including pseudonymized raw data, analysis scripts, and results on a supplementary website: `https://se-sic.github.io/website-event-analysis/` and `https://zenodo.org/records/10614399`.

Of course, the results of our explorative empirical study are not yet suited to be used in practice, since the hypotheses that we develop need to be evaluated by additional studies, ideally at large scale. Yet, with our preliminary investigations and by devising hypotheses, we open the door for future investigations in this direction that can follow-up on our investigations.

7.1 Background & Related Work

Before we describe our methodology, we first provide background information and related work that addresses organizational events in OSS projects as well as the success or failure of an OSS project. At the end of this section, we provide an overview of related studies that either pursue a similar objective to our study or that use a similar notion of organizational events to investigate various aspects of project sustainability.

7.1.1 Organizational Events

During the course of an OSS project, specific events can happen that may affect the evolution of the project. In this study, we consider different kinds of organizational events, namely (1) the emergence of forks, (2) the take over or withdrawal of an OSS project by a commercial company, (3) the abandonment of individual developers who had maintainer roles in the project, (4) and other technical and organizational changes in the project (e.g., a change of the version-control system or a license change). In the following, we provide general background information on the different kinds of events and how they can affect the social and organizational structure of OSS projects.

The Emergence of Forks Fogel defined a *fork* as a "copy of the source code" that can be used "to start a competing project" [Fog05]. Similarly, Nyman et al. [NML+12] characterized a fork of a software project as a "version which seeks to continue to exist apart from the original". While the possibility to fork a project is generally seen as a positive aspect of software development since it allows replicability, customizability, and extendability of a project, it is also seen as a threat to the sustainability of a software project [Fog05; NML+12]. In case of a strong deviation between a fork and the original project, there is a risk that users and developers leave the original project (i.e., they stop using it and stop contributing to it) and move to the fork instead [Fog05; NML+12].

However, not every fork of a software project leads to a migration of users and developers, as there are different types of forks that fulfill different purposes. Zhou et al. [ZVK20] distinguish between *hard forks* and *social forks*, which show significant differences in their amount of activity [HGF+22]. While hard forks lead to a splitting of the project into separate development communities [ZVK19], social forks are usually created to temporarily develop new features in a fast and efficient way, often with the goal to merge them, in the long run, back into the original project [ZVK20]. Especially since the emergence of social coding platforms, such as GitHub, forks are frequently used as a means of distributed coding, in which developers independently work on their own fork and submit pull requests (PRs) to the original project for integrating their code changes into the original project [DST+12; GPD14; KGB+16; ZVK20]. Therefore, in our study, we do not consider the creation of social forks as organizational events, as this type of forks is part of the standard contribution model of social coding platforms [JLH+17]. Consequently, when we talk about the emergence of forks as organizational events, we always refer to hard forks. Nevertheless, Zhou et al. [ZVK20] found that it is not trivial to distinguish between social forks and hard forks. In particular, nowadays, hard forks occur only rarely, but if they occur, they are potentially costly for the project's community [ZVK19].

A substantial corpus of research has already investigated the reasons and motivations why hard forks emerge. One reason could be the intention to provide different variants of a single software product for different hardware or different use cases [FLL+14; RBB+22; ZVK20]. Such forks are often maintained together and are also called a "software family" [BON+22], since the original project and its forks only deviate slightly to serve slightly different purposes. Hence, "software families" are also not considered in our study. Another motivation for creating a fork could be that the original project is or seems to be abandoned, and a new development team (may it be only a single developer or multiple developers) tries to continue the unmaintained or abandoned project [NM11; RG12; ZVK20]. According to Zimmermann [Zim20], such forks are called "friendly forks". Furthermore, sometimes there are legal reasons for forks (e.g., changes in licenses), or companies create their own forks of a project to make it compatible with their closed-source features or to build a branded version of it [RG12; ZVK20]. Finally, hard forks can also be motivated by disputes (of technical or inter-personal nature) among developers [GL14; NM11; RG12; ZVK20]. Interestingly, hard forks are often not created intentionally, but start as soft forks, having the goal to merge changes back to the original project. For example, if developers receive answers to their pull requests too slowly, or if their pull requests are rejected, or if they face technical or personal issues (e.g., issues with external libraries, or features that are undesired by the maintainers of the original project), developers continue working on their soft forks, eventually turning them into hard forks [ZVK20]. When the hard forks gain more and more supporters, who also desire these technical solutions, hard forks often change their name to highlight that the fork has diverged from the original project [ZVK20]. Sometimes, the emergence of a hard fork is even community driven, for instance, when a substantial fraction of the developer community does not agree with the decisions of the maintainer of the original project [RG12].

The emergence of hard forks can have different effects on the original project. According to a study by Zhou et al. [ZVK20], in most cases in which an active project had been forked, no interaction between the fork and its origin happened. Even more, in 47.6% of their investigated cases, the forks live longer than the original project, whereas in 43.6%, the original project outlived the fork. Only in 8.8% of the cases, both, the original project and the fork, stayed active. When an inactive project had been forked, more than 60% of the forks were successful, and in 5% of the investigated inactive projects, the original project even became active again after the fork had emerged [ZVK20].

All in all, as demonstrated by the previous research presented above, the emergence of a hard fork can drastically influence the reputation, the developer behavior, and, in the worst case, even the existence of the original project.

The Role of Companies As we have already seen above, the involvement of companies can lead to forks. However, companies not only create their own forks of OSS projects to satisfy their own requirements on the project [ZVK20], but can also participate in the original project without forking it. Nevertheless, the involvement of a company in the original project can also cause organizational changes. For example, when the Oracle Corporation took over Sun Microsystems, which was the owner of OpenOffice, Oracle restricted the community participation of OpenOffice (e.g., by forcing developers to sign a new contributor licensing agreement (CLA)), which discontented and frustrated the developers and eventually led to the emergence of LibreOffice as a separate fork [GL14; MCD19]. Although we have already considered the emergence of forks as a kind of organizational events, we treat the new involvement of a company in an OSS project as a separate kind of organizational events, which not always lead to forks, but also can affect the developer participation (e.g., loss of developers or attraction of new developers) directly without the emergence of forks. In fact, the role of companies in OSS is different, depending on the intention of the company and the underlying governance structures of a particular OSS project [ZSL+22]. While some companies accept the existing governance structures and only contribute to the OSS project for a limited amount of time to get a specific feature implemented, other companies want to gain strategic influence in the project and plant their employees as paid core developers in the OSS project [BGL+18; BGL+21; Lee12]. This, again, can influence the participation of voluntary developers [HR17; ZMM+16]. Also when companies cease the support of an OSS project, that can influence the developer participation, either leading to project inactivity in consequence of its unmaintained state or to an increase in developer activity due to the end of the domination by the company [ZSL+22]. Not only the organizational structure can be influenced when a company takes over an OSS project or withdraws from an OSS project, but also the software quality can be directly affected. Sato et al. [SWF+13] showed that, in OSS projects that have been taken over by multiple companies one after the other, source-code files that have been changed by multiple companies or organizations contained more bugs than files that have been changed by only one company or organization, respectively.

The Abandonment of Developers Beside the consequences of forks and the involvement of companies, there are also other (possibly unknown) reasons why developers leave a project. According to Marsan et al. [MTM+19], the "loss of contributors" is "one of the most important social problems"[MCD19; MTM+19] in OSS projects. In their interview study, they identified multiple factors that can cause a loss of contributors: On the one hand, conflicts between developers, but also toxic and aggressive communication behavior (especially from maintainers) can cause developers to stop contributing to the project. On the other hand, a general loss of motivation or the emergence of other projects that are more attractive to a specific developer may make them leave the project [MTM+19]. Miller et al. [MWK+19] found by means of a survey that also occupational reasons (such as "getting a new job") are often mentioned as the main reason for quitting project engagement, among other social reasons (such as "personal circumstances").

The reasons and consequences of developer turnover have already been investigated in previous research [e.g., FPB+15; FSV20; JYF24; LRS17; RG06; RGH09; SHD12]. Mockus [Moc10] found that the abandonment of developers can cause "gaps in knowledge" that may affect software quality negatively and increase the defect-proneness of a project. The issue-fixing time tends to be longer, accompanied by slower implementation progress, in projects with a high rate of leaving core developers in contrast to stable projects [FSV20]. Izquierdo-Cortazar et al. [IRO+09] showed that the amount of knowledge loss in case of leaving core developers is project dependent: It ranges from projects that continue as without any developer turnover to projects that rewrite parts of the source code with a new developer team. In the latter case, the rewriting has become necessary because the needed knowledge about these parts of the source code has been lost after the respective developer had left the project. Moreover, Izquierdo-Cortazar et al. [IRO+09] provide measures to mitigate the risks of knowledge loss by recommending that multiple core developers should work on each part of the project, to enforce that multiple developers acquire knowledge on these parts.

Closely related to the abandonment of developers and the corresponding knowledge loss is the concept of the "truck factor". Williams and Kessler [WK03] introduced the truck factor as "the number of people on your team that have to be hit by a truck [...] before the project is in serious trouble". Thus, a low truck factor in a project indicates a serious risk of knowledge loss if one or few developers suddenly abandon or are not available any more, for instance, due to an unexpected death [APH+16]. Although there are different computation algorithms for the truck factor [e.g., APH+16; FMV+19; RM10; RZD+16], the general concept of the truck factor helps quantify the risk of knowledge loss in consequence of the abandonment of central and important developers. Therefore, if core developers leave a software project, we consider this an organizational event that can heavily impact the organizational structure of the project.

Other Organizational Events Finally, there are also other organizational events that belong to neither of the above described kinds of events. For example, a change of the project's main communication channel (e.g., switching issue trackers) may affect the organizational structure of a software project and may influence the project's success [ZC19]. Also, changes in the used contribution technologies (e.g., a switch from the version-control system SVN to GIT) can lead to developers struggling with the new technologies or complaining about them and, in turn, affect project sustainability [AS09; MBN+14; RA12; SH17]. Moreover, even license changes can affect the contribution behavior of developers, leading to project failure if a license is too restrictive, for instance [MS20; SH17; VR15]. Furthermore, also scheduled events such as release dates, at which a new revision of the software product is planned to be rolled out, can affect the organizational structure of an OSS project, since developers usually show different amounts of activity with respect to different development phases in a release cycle [ACM+21; KAD+15]. Thus, the list of organizational events that may potentially influence the organizational structure, the developers' contribution behavior, or the projects' sustainability could be continued endlessly. Consequently, our division of organizational events into different kinds of events is not exhaustive, and the list of events that we consider in our study is only an exemplary selection from the variety of organizational events that can potentially occur in an OSS project.

7.1.2 Success and Failure of Open-Source Software Projects

Closely related to the occurrence of organizational events is also their influence on the success of an OSS project, as already indicated above while describing the different kinds of organizational events. As our motivation for the investigation of organizational events is driven by the potentially negative consequences of specific organizational events that need to be avoided—in the worst case, the abandonment or failure of the project—we provide, in what follows, an overview of different notions of success and failure of OSS projects and how OSS projects are evaluated with respect to success or failure.

Reasons for Project Failure In the literature, the success and failure of OSS projects has been investigated from different points of view, using different measures. While Trinkenreich et al. [TGW+22] investigated the success of individual developers and their contributions in OSS projects, which often is perceived subjectively, we instead look at the success or failure of a project itself. Ewusi-Mensah and Przasnyski [EP91] found that, when projects are abandoned, behavioral and organizational issues usually play a more significant role than pure technical issues. Coelho and Valente [CV17] conducted a survey among maintainers of abandoned OSS projects (i.e., projects without any commits during the last year) and collected nine reasons for project failure (which they equate with project abandonment). The majority of projects have been abandoned because the project itself was considered "obsolete" or competitor projects (that were often initiated or dominated by commercial companies) have emerged and have become more popular, or companies even took over the project. Besides, "outdated technologies" and maintainability problems were mentioned as reasons for project failure. Also social issues, such as the lack of time or interest or conflicts with other developers, are seen as reasons for the failure of a project. Finally, legal problems (e.g., caused by licensing issues) and also the lack of expertise are considered to be causes for project failure [CV17]. To mitigate the risk of project failure, their survey participants proposed to improve the attractiveness of a project to attract new developers, and also to admit additional core developers and maintainers to which the project can be transferred if the initial maintainers are unable to deal with the amount of maintenance work or decide to abandon [CV17]. Also, Fang et al. [FHV23] discussed that project attraction is important for project success.

Raja and Tretter [RT12] identified failed OSS projects as projects on the platform SourceForge that are labeled as inactive and showed that different measures, such as the frequency of software releases, the reaction time on reported bugs, or the rate of problems that are addressed by appropriate specialists can be used to differentiate between failed and successful projects. Further research has shown that restrictive licenses and periods of inactivity or reduced activity threaten the success of an OSS project [CMP07; SAM06; SSN09]. Also, an increasing number of developers is beneficial for project success, whereas a too large number of developers can also threaten the success if coordination problems arise [CMP07].

Project Success and the Characteristics of Developer Networks Joblin and Apel [JA22] investigated whether socio-technical aspects are related to project failure or project success and used simple network characteristics (such as vertex degree or clustering coefficient) to predict the success or failure of an OSS project based on the developer activity in early stages of a project. As in previous studies, they defined failed projects as abandoned projects. Notably, their approach was able to generalize to unseen projects (which were not involved in the training process of their prediction model). Comparing the prediction performance of developer networks (in which developers are connected with each other when they contribute to the same function) to the prediction performance of developer-artifact networks (in which developers are connected to the artifacts, that is, functions, they have worked on) revealed that developer-artifact networks had a higher prediction performance in their study. In addition, they also were able to identify early indicators of project success: Developers need to have comparably low degrees, meaning that they only have few other developers they are collaborating with, but they need to have a high local clustering coefficient, meaning that the developers they are collaborating with are also collaborating with each other. This way, coordination processes are easily satisfiable and developers work in groups that are densely connected and concentrate on similar parts of the source code. Moreover, stable groups of developers that work on the same artifacts, having a low amount of developer turnover, seem to be beneficial for project success [JA22].

Recently, researchers tried to examine whether developer communication related to governance policies in OSS projects is associated with project success. Yin et al. [YCY+22] found that failed and successful projects show different patterns of communication activity. In addition, Yin et al. [YZF23] investigated organizational changes in Apache Incubator[4] projects, in which every project that leaves the incubator is labeled as "graduated (sustainable)" or "retired (unsustainable)" by committees of the Apache Software Foundation. Thus, "retired (unsustainable)" projects can be seen as failed projects, whereas "graduated (sustainable)" are successful projects. For their study, they constructed developer networks based on e-mail communication as well as on cochange information, using ranges of 1 month, and analyzed episodic (i.e., untypical, sporadic) changes of specific network characteristics (e.g., average degree, network average clustering coefficient, or density). They found that successful projects have shorter intervals of organizational changes than failed projects, and the developers in successful

4 The Apache Software Foundation helps auspicious and innovative software projects, usually when they stand at their beginning, to grow and progress within the Apache Incubator by providing services, mentoring, and technical infrastructure as well as guiding them on applying the governance policies of the Apache Software Foundation, as stated on their official website: https://incubator.apache.org/ (accessed at 2023-11-20)

projects generally show more engagement and respond faster than in failed projects [YZF23]. Although they investigate similar network characteristics than we do and search for significant changes in these characteristics, they do neither investigate these changes at specific points in time, nor do they relate identified changes in these characteristics to specific events.

7.1.3 Studying the Role of Organizational Events in the Literature

Having introduced general background information on organizational events and the success or failure of OSS projects above, we now present related work on investigating the role of organizational events in OSS projects.

First of all, let us distinguish organizational events, which we use in our study, from other kinds of events in OSS projects. There are studies that investigate other kinds of events, for instance, activity events in issues (i.e., commented events, see also Chapter 6) to identify patterns in sequences of developers' activities [e.g., DR21; RD15]. Li et al. [LCQ21] considered events as the heaped occurrence of negative comments in issue discussions and grouped them into specific categories (e.g., disagreement or third-party problems). Although analyzing the activities of developers can be helpful to identify factors for project success or failure, we aim at analyzing how specific, sometimes even drastic, organizational events that happen at a specific point in time (e.g., external events such as company acquisitions, as well as internal events such as developer abandonment) affect the network structure of an OSS project.

Other research focused on identifying organizational events in OSS projects from a pure technical perspective, whereas we analyze how these events influence the interaction between developers. For example, Yilek et al. [YRS+09] considered a severe security vulnerability in the Debian OpenSSL package that was identified in 2008. In their study, they investigated how long it took until the vulnerability was fixed on web servers all over the world and how certificate authorities dealt with servers that have not been fixed, but they did neither consider social aspects nor developer interactions. In a similar vein, Durumeric et al. [DLK+14] investigated how the Heartbleed[5] vulnerability in OpenSSL, which was detected in 2014, has been fixed. They monitored how and how fast the corresponding software patches have been applied to web servers, and they analyzed the notification behavior about the vulnerability and its technical consequences.

Also investigating the Heartbleed vulnerability, but closer to our study, Walden analyzed the organizational consequences of Heartbleed [Wal20]. That is, they analyzed how the number of developers, the number of commits, the number of lines of code, and general software engineering practices have changed after Heartbleed. Regarding the analysis of software engineering practices, they studied information on web sites, mailing lists, and the source-code repository and interviewed a central developer who joined OpenSSL shortly after Heartbleed to support the community in changing the processes. In their study, they not only considered the detection of Heartbleed, but also other organizational events later on (e.g., the announcement of new coding guidelines, or the publication of a general code cleanup). In particular, they compared several code and project characteristics in the 25 months prior to and the 25 months after each of the events (based on visual observations in plots, but also statistically confirmed by using regression discontinuity models) as well as across

5 "The Heartbleed Bug": `https://heartbleed.com/` (accessed at 2023-11-30)

multiple releases on the complete evolution time period of the project. Overall, they identified substantial improvements in OpenSSL after the detection of Heartbleed, regarding code quality as well as regarding development processes. For example, OpenSSL established GitHub issues as an issue tracker and changed their release strategy after Heartbleed. While OpenSSL was a largely inactive project prior to Heartbleed, the developer activity in the project has steadily increased thereafter. Thus, according to Walden [Wal20], the project has turned into a sustainable project after all the organizational changes have been applied in consequence of the severe Heartbleed event.

Karus transformed time series of descriptive project statistics (e.g., number of active developers) into wavelets and applied filters afterwards [Kar13]. Then, they checked for anomalies in these wavelet-representations of time series and identified evolutionary patterns, that is, points in time were a specific event must have happened that has led to anomalies in different project statistics. In contrast to our study, they only look at changes in the statistics to detect possible dates for events without relating them to documented events, whereas we search for documented events and investigate whether we can find anomalies in the statistics around the point in time at which the event happened.

Similar to our study, also other studies detected organizational events in OSS projects that could have an influence on the developer activity or organizational structure in the project. Ververs et al. [VBJ11] collected 116 events from websites related to the Debian project and grouped them into 15 categories of events (e.g., releases, incidents, major bugs, conferences related to Debian taking place, etc.). Then, they investigated whether changes in the number of developers or in the average number of commits per developer are correlated with the occurrence of these events. As multiple events occurred temporally close-by to each other, they combined multiple of the events into one event, as otherwise it would be difficult to associate observed changes over time to any of the individual events. Particularly, they also considered changes that occurred up to four weeks before or after an event, to account for time-shifted consequences. In summary, they found that only in less than 11% of the events there was a correlation between the event and a change in developer activity. The most influential events in this regard were dependency issues, problems with releases, or general incidents.

Shaikh and Henfridsson [SH17] extracted 10 events (mostly related to technology changes, such as the change of the used version-control system) in early phases of the Linux kernel project, based on manual inspections of the discussions on the corresponding mailing list. Then, they analyzed the evolution of the project's governance by characterizing the different coordination processes that were used in the time periods between the events. Different from our study, they investigated only the contents of selected e-mail discussions from a theoretical point of view but did not analyze any practical collaboration or communication data.

To focus on company involvement and the long-term sustainability of forks, Gamalielsson and Lundell [GL14] investigated the evolution history of OpenOffice, LibreOffice, and Apache OpenOffice, which all originated from the same project. In particular, they analyzed how developer activity and developer retention evolve with respect to the emergence and co-existence of forks and how knowledge is transferred among the three interrelated projects. While we investigate similar events, our focus is on how such events affect the organizational structure and particular characteristics of developer networks.

Zanetti et al. [ZST+13] analyzed the bug tracker of project Gentoo and identified that, starting at a certain point in time, a single central developer was involved in most of the interactions with other developers. However, when this developer had left the project unexpectedly, the project's developer community was struggling with problems. To investigate the consequences of the developer's abandonment on the organizational structure as well as on the bug handling performance, Zanetti et al. [ZST+13] constructed developer networks for 30-days time ranges based on assigned–assignee relationships on bugs, using a sliding-window approach, and analyzed how the characteristics of the developer network (centrality, clustering coefficient, connectivity) have changed over time. As long as the considered developer was active in the project, the networks had a high clustering coefficient as all others were centered around this one developer, but when this developer left the project, the network became sparse and had only a small clustering coefficient. In our study, we use a similar approach using slightly different notions and time ranges of developer networks, investigating an even more diverse set of different network characteristics on a variety of different kinds of events.

As different interaction channels are involved in OSS projects, it is possible that some events influence only some of the channels. Therefore, Huang et al. [HCQ+22] developed an event-monitoring model which helps to identify organizational events on different interaction channels and data sources. Also in our study on organizational events, we investigate different interaction channels and examine whether and how the characteristics of collaboration networks and communication networks are affected by organizational events.

7.2 Methodology

With our exploratory empirical study, we want to find out whether there are specific network characteristics that could be used as indicators for specific kinds of organizational events (see Section 7.1.1 for the different kinds of events). Thus, we ask the following research question:

> **RQ:** Do particular network characteristics reflect changes in the network structure when specific organizational events (such as fork creation, company acquisition, the abandonment of core developers, etc.) happen?

To answer our research question, we take publicly known organizational events in specific OSS projects and investigate whether and how particular network characteristics have changed in their values around the points in time at which the events happened. This way, we aim at identifying network characteristics that might be sensitive to such organizational events.

In the following, we provide information on our data-extraction procedure, we describe which network characteristics we investigate, and how we identify whether changes in specific network characteristics could be related to organizational events.

7.2.1 Data Extraction

For all the projects that we analyze in our case studies, we use Codeface to gather commit data from the corresponding Git repository. Depending on whether a project has used mailing lists or GitHub issues as communication channel for developers, we extract the corresponding e-mail data from the publicly available mailing-list archive Gmane or the corresponding issue

Table 7.1: Overview of the network characteristics that we investigate with respect to the occurrence of organizational events.

Network Characteristic	Description
Average degree	Captures with how many other developers a developer interacts on average.
Average eigenvector centrality	Provides an overview of whether there are many developers that have a high centrality or only few ones.
Average path length	Indicates how many developers, on average, need to pairwisely interact with each other to pass information from one developer to any other developer in the network.
Density	Is a general indicator of the level of activity (i.e., the interactions between the developers) in the project.
Global clustering coefficient	Indicates whether the network contains tightly connected clusters or rather loosely connected individuals.
Network average clustering coefficient	Indicates whether the developers, on average, are tightly connected with many others and, thus, well integrated into the project community.
Hierarchy	Indicates whether the project has "a centralized governance structure".
Modularity	Indicates whether there is a strong community structure in the network.
Scale-freeness	Indicates that a network is robust to changes.
Small-worldness	Indicates a fast information flow due to small average path length and high clustering coefficient.

discussions from GitHub using our tool GitHubWrapper, as described in Section 2.4. For projects that have used both communication channels extensively (independent of whether they have been used one after the other or in parallel), we extract data from both of them.

In addition, we extract organizational events and the dates at which they have happened via an extensive web search in blog posts, news articles, issue discussions, or on the project's website. We provide more information on that when we describe the specific events that we have extracted for our case studies (see Section 7.3.1).

7.2.2 Obtaining Network Characteristics in Relation to Events

From the obtained project data, we build undirected, simplified developer networks using the network library coronet, as explained in Section 2.3.1. In particular, we build cochange networks as well as communication networks (i.e., mail networks or issue networks), to account for both, the organizational structure behind the source-code changes and the organizational structure behind the communication activities. Similar to the methodology that we have used in Chapters 5 and 6, we split the data into overlapping time ranges of 6 months [HKC+11; MW11; PBD+14], using the sliding-window approach that we have introduced in Section 2.3.2.[6] For each of the 6-months networks, we extract various established network characteristics, similar to previous studies that analyzed the evolution of developer networks [e.g., JAM17; Job17]. In Table 7.1, we provide an overview of the network characteristics that we use in our study; all of them have already been explained in Section 2.3.3. To

6 In a sensitivity analysis, we also tried time ranges of 3, 9, and 12 months. While 9-months ranges yield similar results to 6-months ranges, 3-months ranges are too noisy, and 12-months ranges only seldomly reveal changes.

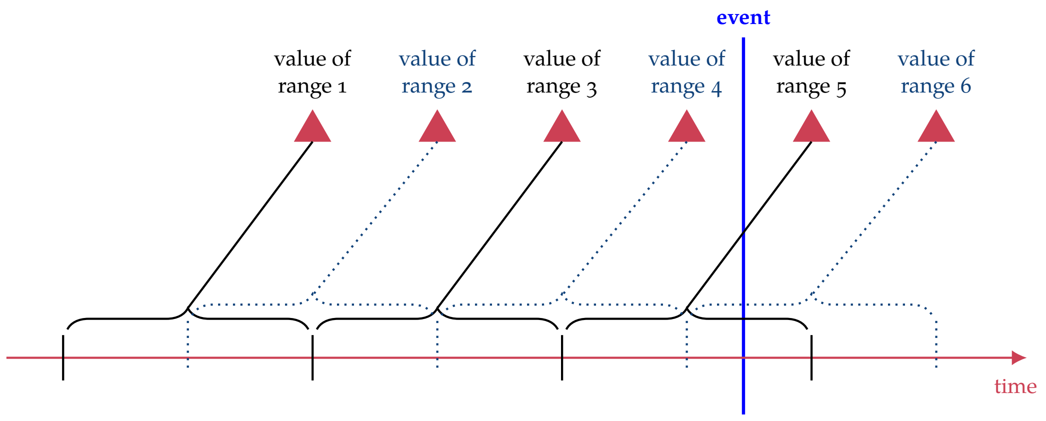


Figure 7.1: Exemplary visualization of how we visualize the results of our study for the evolution of a single network characteristic of a single project over time: For each time range, we plot the value of the investigated network characteristic (here as a triangle, but we also use other shapes in our result plots). The values can be different for each time range, but we neglect this here in the exemplary visualization for simplicity reasons. As one value of the network characteristic represents not only a single point in time, but the network of a complete time range, we highlight the corresponding time range in this exemplary visualization. From that, we can see that the point in time at which the actual value is depicted corresponds to the end of the time range it represents. This is reasonable, as the network of a time range captures all the activities up to the end of the time range. The blue vertical line represents an organizational event that happened at a certain point in time. Note that the event can occur within any time range, *not* necessarily at the beginning or end of a time range.

account for general changes in the amount of activity in a project, independent of the network structure, we also determine the *number of developers*, the *number of commits*, the *number of e-mails*, and the *number of issue comments* for each time range.

For each of the extracted network characteristics and numbers, we plot their respective values over time. On the time line, each value refers to the end of the respective 6-months window, to depict that the corresponding value considers the activities up until that point in time at which we show it in the plot. In Figure 7.1, we provide an exemplary visualization of how we plot the evolution of a specific network characteristic over time. In addition to plotting the respective values for each time range, we also visualize the occurrence of an organizational event by a vertical line at that point in time at which the event happened. Note that the events typically do not fall on the same date as the end of our time ranges do. Thus, the events usually happen within one of the 6-months time ranges for which we depict the corresponding values of the respective network characteristic, as we demonstrate in our exemplary visualization in Figure 7.1. Also note that one value of a network characteristic for a specific time range only belongs to one specific network type. That is, we depict different values for different network types (e.g., the cochange network and the issue network) for the same time range using different shapes and colors.

7.2.3 Assessing the Influence of Events on Network Characteristics

To assess whether a specific network characteristic may potentially indicate changes in the organizational structure of an OSS project, we investigate whether we can spot any differences in the respective network metric before and after the event happened. As the event does not necessarily separate two time ranges, we cannot make a clear cut between time ranges before and after an event has happened. Nevertheless, this is not desirable, though, as we do not expect that organizational changes affect the network structure suddenly. While there may be events that might be foreseeable and show themselves in gradual changes to the network structure until the actual event happens (e.g., longstanding core developers who plan to leave the project might already become inactive or less active in the months or weeks before they actually leave or announce to leave), other events might affect the network structure only in the ranges after the event (e.g., if a core developer passes away in an accident, this cannot be visible before the event happens). Therefore, we look for general changes in the network characteristic that are clearly visible, independent of whether they happen close to the event or shortly before or after it.

Notice that with our exploratory empirical study, we aim at investigating whether we can find changes in the network structure that might be indicators for organizational events. Therefore, due to the preliminary nature of our study, we do not apply any statistical methods to determine significant differences. Instead, we look for clearly visible, evident changes in the network characteristics (similar to the initial observations of changes in project characteristics in the study of Walden [Wal20]), without questioning their significance from a statistical perspective. As we consider our study to be a preliminary study on whether developer-network characteristics reflect organizational events, we leave any statistical checks open to future studies which need to evaluate our results at large-scale using appropriate statistical methods. Consequently, as a result of our study, we devise hypotheses based on our visual insights. These hypotheses shall serve as entry points for future studies in which the hypotheses need to be evaluated. For more details on the different possibilities of how our study needs to be continued, we refer to Sections 7.4 and 7.5.

7.3 Exploratory Study

In our exploratory empirical study, we investigate 53 selected organizational events of various kinds in 8 different case studies. A *case study*, the way we use the term in this study, either consists of a single OSS project, or of multiple OSS projects that have a common history (e.g., one project being a fork or the continuation of the other project). We start with a brief description of our case studies and the various events that we investigate in a particular case study. Thereafter, we present our observations from analyzing the different, above mentioned network characteristics with regard to potential changes around the points in time at which an organizational event has occurred.

7.3.1 Case Studies and Events

For this study, we have selected well-known, long-standing projects as our case studies, for which we were able to extract events from official websites, news articles, or blogs, etc. In what follows, we briefly describe each of our case studies and summarize the corresponding events. As already stated above, all the information about the events has been collected from blog posts, news articles, the projects' websites, or similar sources on the Internet. We provide references to the respective sources of information for each event in Table 7.2, in which we provide a comprehensive overview of all the organizational events that we investigate in our study. Note that we assign a unique identifier $\mathrm{E}x_y$ to each event, where x denotes the number of the case study in which the event occurs and y denotes the number of the specific event. Within each case study, the events are temporally ordered.

ownCloud & Nextcloud Our first case study consists of the projects ownCloud and Nextcloud, which are cloud platforms for sharing files. In 2016, the founder and main maintainer of ownCloud resigned and left the project, which represents our event $\mathrm{E1}_1$. Five weeks later, Nextcloud was created as a fork of ownCloud ($\mathrm{E1}_2$). Another six weeks after that, a new CEO and new financial investors joined ownCloud ($\mathrm{E1}_3$).

Gogs & Gitea In our second case study, we look at the projects Gogs and Gitea. Gogs is a hosting service for Git repositories written in the programming language Go. In 2015, the main developer of Gogs was unresponsive (i.e., he did not respond to any comments and did not merge any pull request, etc.) and disappeared for more than a month ($\mathrm{E2}_1$). As a consequence, the Gogs community continued the development on a new fork called Gitea ($\mathrm{E2}_2$). Even after the main developer came back, he announced that he will not merge the changes from Gitea back into Gogs, which led to an eventual divergence of the two projects ($\mathrm{E2}_3$).

htop Third, and similarly to the previous case study, we have a look at project htop, which is a process manager for Linux environments. In February 2019, the original creator of the project became inactive in the project ($\mathrm{E3}_1$). As he did not react to anything in the project up until August 2020, the community wondered whether he still maintains the project. Consequently, a team of developers decided to continue the project in a new repository and created a fork of htop, also migrating all the open pull requests to the new fork ($\mathrm{E3}_2$). Finally, in September 2020, the maintainer of the original project informed the community that he did not have a look at the project since his last activity and that he has thought about handing over the maintainership to someone else. Nevertheless, he noticed that this was no longer necessary as a new team has continued the project in a new fork anyway. Thus, he officially announced that the project is continued in the new fork ($\mathrm{E3}_3$).

Node.js Our fourth case study is project Node.js, which is a JavaScript runtime environment. As, in 2014, a substantial group of developers was dissatisfied with the governance structure of the project (e.g., how the community can solve problems efficiently), they created io.js as a fork of Node.js ($\mathrm{E4}_1$). After intensive discussions between the leaders of the original project and the fork, roughly half a year later, they agreed on changes to the governance structure in Node.js and decided to cooperate again ($\mathrm{E4}_2$). However, a little over two years later, again a new fork of Node.js was created, namely Ayo.js, as there were ongoing, controversial debates in Node.js regarding potential behavioral code-of-conduct violations by specific developers ($\mathrm{E4}_3$). Nevertheless, also this time, the new fork was discontinued 1 year later ($\mathrm{E4}_4$).

Qt As our fifth case study, we investigate project Qt, which is a toolkit for the development of graphical user interfaces. Qt had started as a primarily commercial project and switched to a fully open governance model in 2011 ($E5_1$). During the course of the existence of Qt, it belonged to different companies, which restructured or closed some of its departments ($E5_2$) or divested Qt to other companies ($E5_3$, $E5_4$).

MySQL & MariaDB The database management systems MySQL and MariaDB form our sixth case study. First, Sun snapped MySQL ($E6_1$, $E6_2$) in 2008, and, less than 2 years later, Oracle, in turn, bought Sun ($E6_5$, $E6_7$). In between, one of the co-founders of MySQL left the project and created, together with colleagues, MariaDB as a fork of MySQL ($E6_4$, $E6_6$). Later on, a MariaDB foundation was established and merged with another foundation ($E6_9$, $E6_{10}$, and $E6_{12}$), leading to a stabilized coexistence of MySQL and MariaDB. Moreover, also another famous developer of MySQL left the project ($E6_8$). Beside all these company-related or developer-abandonment-related events, also changes of the version-control system ($E6_3$, $E6_{13}$) or communication channel ($E6_{11}$) occurred in MySQL.

OpenOffice & LibreOffice & Apache OpenOffice In our seventh case study, we look at the office software OpenOffice and its successors LibreOffice and Apache OpenOffice [GL14]. As OpenOffice was owned by Sun, also in this case study the acquisition of Sun by Oracle is present ($E7_1$, $E7_2$). Since a substantial part of the OpenOffice community assumed that Oracle is not interested to continue in the long run, LibreOffice was created as a fork of OpenOffice in 2010 ($E7_3$). Noteworthy, the Brazilian government decided to support the development of both OpenOffice and LibreOffice ($E7_6$). Just over six months after the creation of LibreOffice, Oracle indeed announced that it will discontinue OpenOffice ($E7_4$) and donated the project to the Apache Software Foundation ($E7_5$), which made a license change and continued the project under the name Apache OpenOffice ($E7_7$). First, IBM actively participated in the development of Apache OpenOffice, but then silently withdrew from the project in 2012 ($E7_8$). Then, in 2016, discussions about a potential retirement of Apache OpenOffice started ($E7_9$). In 2019, Apache OpenOffice changed its version-control system ($E7_{10}$). Eventually, in 2021, a critical security vulnerability in Apache OpenOffice became known ($E7_{11}$).

OpenSSL A critical security vulnerability is also relevant in our last case study OpenSSL, which is a widely-used encryption library. In 2014, OpenSSL made headlines with the serious Heartbleed vulnerability ($E8_1$). As a consequence of this vulnerability, comprehensive changes in the organizational processes of OpenSSL (such as changes to the release strategy, coding style, and code base, but also license changes and changes of the communication channel or the introduction of continuous integration) were carried out later on ($E8_2$–$E8_7$) [Wal20]. In 2017, OpenSSL introduced a specific management committee ($E8_8$). Later on, co-founders of OpenSSL left the project ($E8_9$), while new members of the management committee and new committers have been appointed ($E8_{10}$, $E8_{11}$).

For more details on all the individual events in the different case studies, we refer to Table 7.2 and the corresponding footnotes.

Table 7.2: Overview of the different organizational events that we investigate in our study.

Case Study	Event	Description	Occurrence
ownCloud & Nextcloud	$E1_1$	Resignment of the founder of ownCloud[7]	2016-04-27
	$E1_2$	Start of Nextcloud as a fork of ownCloud[8]	2016-06-02
	$E1_3$	New CEO and finance investments for ownCloud[9]	2016-07-14
Gogs & Gitea	$E2_1$	Disappearance of the main developer of Gogs[10]	2015-06-01
	$E2_2$	Start of Gitea as an official fork of Gogs[10]	2015-06-30
	$E2_3$	Comeback of the main developer of Gogs & refusal to merge changes from Gitea[10]	2015-07-24
htop	$E3_1$	Start of inactivity of the creator and maintainer of htop (last commit in repository)[11]	2019-02-10
	$E3_2$	Start of migration from the original repository to a new repository[12]	2020-08-17
	$E3_3$	Official statement about project continuation by another team in another repository[13]	2020-09-07
Node.js	$E4_1$	Emergence of io.js fork due to dissatisfaction with the governance structure[14]	2014-12-03
	$E4_2$	Cooperation between io.js and Node.js[15]	2015-05-15
	$E4_3$	Start of Ayo.js as a fork of Node.js caused by behavioral code-of-conduct violations[16]	2017-08-23
	$E4_4$	Abandonment of Ayo.js[17]	2018-08-31
Qt	$E5_1$	Switch to an open-governance model[18]	2011-10-21
	$E5_2$	Announcement that the Qt development department in Brisbane will be closed[19]	2012-08-03
	$E5_3$	Announcement that Digia will acquire Qt from Nokia[20]	2012-08-09
	$E5_4$	Takeover of Qt business and copyrights by Qt company (spin-out from Digia)[21]	2014-09-16

This table is continued on the next page.

7 Frank Karlitschek: “Big changes: I am leaving ownCloud, Inc. today”, 2016: https://karlitschek.de/2016/04/big-changes-i-am-leaving-owncloud-inc-today/ (accessed at 2023-11-30)

8 Frank Karlitschek: “Nextcloud”, 2016: https://karlitschek.de/2016/06/nextcloud/ (accessed at 2023-11-30)

9 ownCloud GmbH: “Press release: ownCloud Secures Financing and Expands its Management Team”, 2016: https://owncloud.com/news/owncloud-secures-financing-expands-management-team/ (accessed at 2023-11-30)

10 GitHub issue: “Is Gogs dead?”, 2015: https://github.com/gogs/gogs/issues/1304/ (accessed at 2023-11-30)

11 Changelog Interviews: “How open source saved htop”, 2020: https://changelog.com/podcast/413/ (accessed at 2023-11-30), and https://github.com/hishamhm/htop/commits/master (accessed at 2023-11-30)

12 GitHub issue: “Is this project maintained?”, 2020: https://github.com/hishamhm/htop/issues/992/ (accessed at 2023-11-30)

13 https://github.com/hishamhm/htop/commit/59166da773d8a9a97c8f1b087346865bc8cda9fa/ (accessed at 2023-11-30)

14 Paul Krill: “Node.js gains a new fork, much to Joyent’s dismay”, 2014: https://infoworld.com/article/2854642/ (accessed at 2023-11-30), and Paul Krill: “Q&A: Why io.js decided to fork Node.js”, 2014: https://infoworld.com/article/2855057/ (accessed at 2023-11-30), and Anand Mani Sankar: “Node.js vs io.js: Why the fork?!?”, 2015: http://anandmanisankar.com/posts/nodejs-iojs-why-the-fork/ (accessed at 2023-11-30)

15 Mike Dolan: “Node.js and io.js leaders are building an open, neutral Node.js Foundation to support the future of the platform”, 2015: https://nodejs.org/blog/community/node-leaders-building-open-neutral-foundation/ (accessed at 2023-11-30)

16 Andrii Lundiak: “Node.js => Ayo.js (again?)”, 2017: https://www.linkedin.com/pulse/nodejs-ayojs-again-andrii-lundiak?trk=pulse-article_more-articles_related-content-card/ (accessed at 2023-11-30), GitHub issue: “Node.js is forked to Ayo.js”, 2017: https://news.ycombinator.com/item?id=15078995 (accessed at 2023-11-30), GitHub issue: “meta: vote regarding Rod’s status on the TSC”, 2017: https://github.com/nodejs/TSC/issues/310/ (accessed at 2023-11-30), and GitHub issue: “what is ayo?”, 2017: https://github.com/ayojs/ayo/issues/125/ (accessed at 2023-11-30)

17 “Ayo.js fork of Node.js quietly shuts down”, 2018: https://news.ycombinator.com/item?id=17977521 (accessed at 2023-11-30), and GitHub issue: “Project status?”, 2018: https://github.com/ayojs/ayo/issues/139/ (accessed at 2023-11-30)

18 Lars Knoll: “The Qt Project is live!”, 2011: https://blog.qt.io/blog/2011/10/21/the-qt-project-is-live/ (accessed at 2023-11-30)

19 Dan Nancarrow: “Nokia closes Australian development office”, 2012: https://www.smh.com.au/technology/nokia-closes-australian-development-office-20120803-23kc4.html (accessed at 2023-11-30)

20 Michael Larabel: “Digia Buys Out Qt From Nokia”, 2012: https://www.phoronix.com/news/MTE1NjY/ (accessed at 2023-11-30)

21 Chris Duckett: “Qt hot potato spun out from Digia into fourth home”, 2014: https://www.zdnet.com/article/qt-hot-potato-spun-out-from-digia-into-fourth-home/ (accessed at 2023-11-30)

Case Study	Event	Description	Occurrence
MySQL & MariaDB	$E6_1$	Announcement that Sun takes over MySQL[22]	2008-01-16
	$E6_2$	Acquisition of MySQL by Sun completed[23]	2008-02-26
	$E6_3$	Migration of MySQL from BitKeeper to Bazaar[24]	2008-06-19
	$E6_4$	Leaving of MySQL co-founder Monty Widenius from Sun[25]	2009-02-05
	$E6_5$	Announcement that Oracle will buy Sun[26]	2009-04-20
	$E6_6$	First release of MariaDB (as a fork of MySQL) by Monty Widenius and colleagues[27]	2009-10-29
	$E6_7$	Completion of the acquisition of Sun by Oracle[28]	2010-01-27
	$E6_8$	Leaving of MySQL developer Zack Urlocker[29]	2010-01-28
	$E6_9$	Announcement that a MariaDB Foundation will be established[30]	2012-12-04
	$E6_{10}$	Merger of SkySQL Ab and Monty Program Ab[31]	2013-04-23
	$E6_{11}$	MySQL joining GitHub[32]	2014-09-26
	$E6_{12}$	Renaming of SkySQL Ab to MariaDB Corporation Ab[33]	2014-10-01
	$E6_{13}$	Migration of MySQL from Bazaar to Git[34]	2015-01-05
OpenOffice & LibreOffice & Apache OpenOffice	$E7_1$	Announcement that Oracle will buy Sun (which owned OpenOffice)[35]	2009-04-20
	$E7_2$	Completion of acquisition of Sun by Oracle[28]	2010-01-27
	$E7_3$	Creation of LibreOffice as a fork of OpenOffice[36]	2010-09-28
	$E7_4$	Announcement that Oracle will stop the development of OpenOffice[37]	2011-04-15
	$E7_5$	Takeover of OpenOffice from Oracle by the Apache Software Foundation[37]	2011-06-14

This table is continued on the next page.

22 Patrick Thibodeau: "MySQL needs us to expand, says Sun CEO Schwartz", 2008: `https://www.computerworld.com/article/2538783/mysql-needs-us-to-expand--says-sun-ceo-schwartz.html` (accessed at 2023-11-30)
23 Paula Rooney: "Sun completes MySQL buy, guns for Microsoft ... but not Oracle?", 2008: `https://www.zdnet.com/article/sun-completes-mysql-buy-guns-for-microsoft-but-not-oracle/` (accessed at 2023-11-30)
24 Kaj Arnö: "Version Control: Thanks, BitKeeper ? Welcome, Bazaar", 2008: `https://planet.mysql.com/entry/?id=13334` (accessed at 2023-11-30)
25 Chris Kanaracus: "MySQL co-founder quits Sun", 2009: `https://infoworld.com/article/2676082/` (accessed at 2023-11-30)
26 Adena Schutzberg: "Oracle Buys Sun: What Happens to MySQL?", 2009: `https://www.directionsmag.com/article/2357/` (accessed at 2023-11-30)
27 "MariaDB 5.1.38 Release Notes", 2009: `https://mariadb.com/kb/en/mariadb-5138-release-notes/` (accessed at 2023-11-30) and XperLab: "What is MariaDB?", 2021: `https://xpertlab.com/what-is-mariadb/` (accessed at 2023-11-30)
28 Alex Handy: "Oracle completes Sun acquisition", 2010: `https://sdtimes.com/mysql/oracle-completes-sun-acquisition/` (accessed at 2023-11-30)
29 Zack Urlocker: "New beginnings for Sun, MySQL – and me", 2010: `https://infoworld.com/article/2629448/` (accessed at 2023-11-30)
30 Rasmus Johansson: "MariaDB Foundation to Safeguard Leading Open Source Database", 2012: `https://mariadb.org/mariadb-foundation-to-safeguard-leading-open-source-database/` (accessed at 2023-11-30)
31 Colin Charles: "SkySQL merges with Monty Program Ab, makers of MariaDB", 2013: `https://mariadb.org/tag/monty-program-ab/` (accessed at 2023-11-30)
32 Yngve Svendsen: "MySQL on GitHub", 2014: `https://dev.mysql.com/blog-archive/mysql-on-github/` (accessed at 2023-11-30)
33 Michael "Monty" Widenius: "Why SkySQL becoming MariaDB Corporation will be good for the MariaDB Foundation", 2014: `https://monty-says.blogspot.com/2014/10/why-skysql-becoming-mariadb-corporation.html` (accessed at 2023-11-30)
34 Yngve Svendsen: "Heads up: Going 100% GitHub at the End of January ", 2015: `https://dev.mysql.com/blog-archive/heads-up-going-100-github-at-the-end-of-january/` (accessed at 2023-11-30)
35 Andrew Brown: "The Sun ain't gonna shine on OpenOffice any more", 2009: `https://www.theguardian.com/technology/2009/apr/30/sun-oracle-openoffice/` (accessed at 2023-11-30)
36 Italo Vignoli: "Celebrating 5 years of LibreOffice", 2015: `https://opensource.com/life/15/9/libreoffice-turns-5/` (accessed at 2023-11-30)
37 Archived News from Apache OpenOffice: `https://www.openoffice.org/news/` (accessed at 2023-11-30)

Case Study	Event	Description	Occurrence
	$E7_6$	Declaration of the Brazilian government to develop OpenOffice & LibreOffice[38]	2011-07-01
	$E7_7$	First release of Apache OpenOffice & license change to Apache License 2.0[39]	2012-05-08
	$E7_8$	Withdrawal of IBM from Apache OpenOffice[40]	2014-08-21
	$E7_9$	Discussion about the potential retirement of Apache OpenOffice[41]	2016-09-01
	$E7_{10}$	Migration of Apache OpenOffice from SVN to Git[42]	2019-08-01
	$E7_{11}$	Disclosure of a serious security vulnerability in Apache OpenOffice[43]	2021-09-18
OpenSSL	$E8_1$	Disclosure of the Heartbleed vulnerability[44]	2014-04-07
	$E8_2$	Begin of a new release strategy[45]	2014-12-23
	$E8_3$	Large changes to the code base & introduction of a new coding style[46]	2015-02-11
	$E8_4$	Completion of code cleanup[47]	2015-07-28
	$E8_5$	Start of using Travis CI for continuous integration[48]	2015-09-03
	$E8_6$	Announcement of transition from RT issue tracking to GitHub issues[49]	2016-10-12
	$E8_7$	License change to Apache License 2.0[50]	2017-03-22
	$E8_8$	Start of OpenSSL management committee[51]	2017-06-13
	$E8_9$	Leaving of two co-founders of OpenSSL[52]	2017-10-24
	$E8_{10}$	Awarding of Levchin Prize to OpenSSL[53]	2018-01-10
	$E8_{11}$	Commencement of new members in management committee & new committers[54]	2018-08-22
	$E8_{12}$	Commencement of new committers[55]	2019-05-20

38 Dj Walker-Morgan: "Brazilian government signs up to develop OpenOffice and LibreOffice", 2011: http://www.h-online.com/open/news/item/Brazilian-government-signs-up-to-develop-OpenOffice-and-LibreOffice-1275068.html (accessed at 2023-11-30)
39 Martin Brinkmann: "Apache OpenOffice 3.4, New Name, New Version", 2012: https://www.ghacks.net/2012/05/08/apache-openoffice-3-4-new-name-new-version/ (accessed at 2023-11-30)
40 Bruce Byfield: "LibreOffice, OpenOffice, and rumors of unification", 2014: https://www.linux-magazine.com/Online/Blogs/Off-the-Beat-Bruce-Byfield-s-Blog/LibreOffice-OpenOffice-and-rumors-of-unification/ (accessed at 2023-11-30)
41 Dennis E. Hamilton: " Contemplating the possible retirement of Apache OpenOffice", 2016: https://lwn.net/Articles/699047/ (accessed at 2023-11-30)
42 https://openoffice.apache.org/source.html (accessed at 2023-11-30)
43 Thomas Claburn: "Apache OpenOffice can be hijacked by malicious documents, fix still in beta", 2021: https://www.theregister.com/2021/09/20/apache_openoffice_rce/ (accessed at 2023-11-30)
44 "The Heartbleed Bug": https://heartbleed.com/ (accessed at 2023-11-30)
45 Matt Caswell: "The New Release Strategy", 2014: https://openssl.org/blog/blog/2014/12/23/the-new-release-strategy/ (accessed at 2023-11-30)
46 Matt Caswell: "Code Reformat Finished", 2015: https://openssl.org/blog/blog/2015/02/11/code-reformat-finished/ (accessed at 2023-11-30)
47 Rich Salz: "Beyond Reformatting: More Code Cleanup", 2015: https://openssl.org/blog/blog/2015/07/28/code-cleanup/ (accessed at 2023-11-30)
48 GitHub issue: "Add initial Travis CI configuration", 2014: https://github.com/openssl/openssl/pull/373/ (accessed at 2023-11-30)
49 Rich Salz: "Face to Face: Goodbye RT, Hello GitHub", 2016: https://openssl.org/blog/blog/2016/10/12/f2f-rt-github/ (accessed at 2023-11-30)
50 Rich Salz: "Licensing Update", 2017: https://openssl.org/blog/blog/2017/03/22/license/ (accessed at 2023-11-30)
51 Tim Hudson: "New Committers", 2017: https://openssl.org/blog/blog/2017/06/13/committers/ (accessed at 2023-11-30)
52 Matt Caswell: "Steve Henson", 2017: https://openssl.org/blog/blog/2017/10/24/steve-henson/ (accessed at 2023-11-30), Tim Hudson: "Steve Marquess", 2017: https://openssl.org/blog/blog/2017/10/27/steve-marquess/ (accessed at 2023-11-30)
53 Matt Caswell: "OpenSSL Wins the Levchin Prize", 2018: https://openssl.org/blog/blog/2018/01/10/levchin/ (accessed at 2023-11-30)
54 Tim Hudson: "New OMC Member and New Committers", 2018: https://openssl.org/blog/blog/2018/08/22/updates/ (accessed at 2023-11-30)
55 Tim Hudson: "New Committers", 2019: https://openssl.org/blog/blog/2019/05/20/committers/ (accessed at 2023-11-30)

Table 7.3: Descriptive statistics of the investigated projects on 6-months ranges and overall.

Project	Commit Authors			Issue Contributors			E-Mail Senders		
	Range min	Range max	Overall	Range min	Range max	Overall	Range min	Range max	Overall
ownCloud	11	91	471	0	1 740	9 975	2	377	1 468
Nextcloud	11	90	626	0	2 158	7 482	—	—	—
Gogs	3	45	292	97	922	5 021	—	—	—
Gitea	25	99	705	0	1 107	6 946	—	—	—
htop	0	35	145	0	189	1 447	—	—	—
Node.js	3	484	2 196	0	2 154	12 061	0	1 145	7 052
Qt	2	181	541	—	—	—	0	346	929
LibreOffice	9	197	1 401	0	6	26	0	496	4 261
Apache OpenOffice	3	28	57	0	16	34	0	526	2 649
MySQL	8	133	420	0	31	223	0	3 622	16 517
MariaDB	8	73	442	0	94	439	—	—	—
OpenSSL	4	75	410	0	610	3 024	0	489	4 834

7.3.2 Observations Regarding Network Characteristics

For each of the above introduced case studies, we investigate each network characteristic that we have listed in Table 7.1 in Section 7.2.2. Due to the sheer amounts of network characteristics, network types, and events, we present only the results for a selection of them in this section. That is, in what follows, we focus on the network characteristics and network types for which we observe the most noticeable changes with respect to the occurrence of organizational events. For the remaining network characteristics, we refer to our supplementary website.[56]

In Table 7.3, we provide an overview of how many developers are active in each of the investigated interaction channels of each investigated project, by providing the minimum and maximum number of developers in a 6-months range, as well as by providing the total number of unique developers that participated in a channel over the complete observation period of a project. This way, we describe the sizes of the different projects.

ownCloud & Nextcloud For project ownCloud, the number of developers as well as the number of activities (i.e., commits, issue comments, and e-mails) reached their maximum during the year before $E1_1$ and started to decrease already before it has happened. After $E1_3$, the strong decrease in the numbers of developers and activities continued for more than a year, while Nextcloud started to grow.

In Figure 7.2, we show the scale-freeness and small-worldness for ownCloud and Nextcloud over time. While the cochange network started to be scale-free only from the sixth time range on, the issue and mail networks were almost always scale-free (ignoring the late time ranges,

56 https://se-sic.github.io/website-event-analysis/

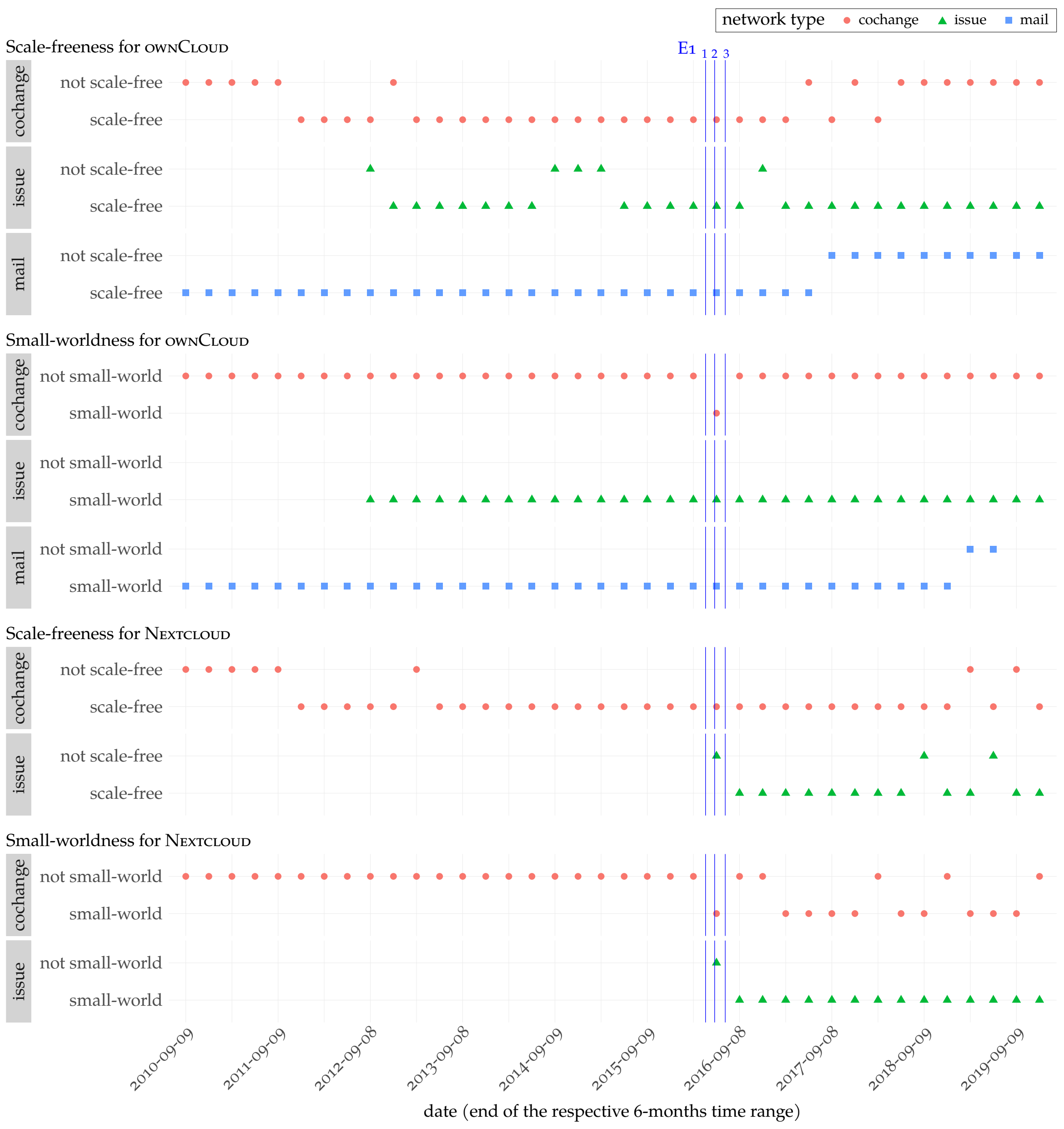


Figure 7.2: Scale-freeness and small-worldness of the cochange, issue, and mail networks for projects ownCloud and Nextcloud over time. The blue vertical lines denote events $E1_1$–$E1_3$.

in which the mail network started to be abandoned). Beginning with the fourth range after $E1_3$, there were fluctuations in the scale-freeness of the cochange network, and also the mail network was not scale-free any more since then. The latter comes along with the starting abandonment of the mailing list for the benefit of GitHub issues. Opposed to ownCloud, the cochange network of Nextcloud consistently stayed scale-free after $E1_3$ (with some exceptions more than 2 years later). Furthermore, the mail and issue networks of ownCloud were always small-world (except for the mail network in late ranges, in which the usage of the mailing list was decreased). This also holds for the issue network of Nextcloud (except for the initial range at fork creation). Noteworthy, the cochange network of ownCloud was never small-world, except for the time range that contains the events $E1_1$ and $E1_2$. Opposed to ownCloud, the cochange network of Nextcloud was, after fork creation, sometimes small-world, sometimes not.

When looking at the other network characteristics, we can see that the average degree for ownCloud has already reached its maximum in the cochange and in the mail network about 2 years before $E1_1$ and was decreasing since then (and also decreased in Nextcloud afterwards). Around the investigated events, we cannot identify noticeable changes. The density in the cochange network started to decrease already 1 year before $E1_1$. Also, the average path length in the cochange network started to decrease before $E1_1$ and started to increase again two time ranges after $E1_3$. In contrast, the average path lengths of the mail and issue networks of ownCloud reached their maximum values around $E1_1$–$E1_3$ or shortly thereafter. The global clustering coefficient of the cochange network reached its minimum around these events and increased again thereafter, while the mail and issue networks already had their minimum values around 1.5 years before these events. The hierarchical structure does not show any noteworthy changes, except for the mail network, in which there was a breakdown in the hierarchical structure around 2 years before $E1_1$, which may be related to ownCloud's shift of communication from the mailing list to the issue tracker.

Gogs & Gitea For projects Gogs and Gitea, the number of developers as well as the number of commit activities stayed largely constant around $E2_1$–$E2_3$. The number of issue comments and issue contributors was growing from the initialization of Gogs, and even continued growing in the year after $E2_3$. Only then, we can observe a decrease when the issues of Gitea started to exist about a year after $E2_3$.

In Figure 7.3, we show the scale-freeness and small-worldness for Gogs and Gitea over time. While the cochange network of Gogs started to become scale-free around 9 months before $E2_1$, it lost its scale-freeness property shortly after $E2_3$ and did not regain the scale-freeness property for the rest of its evolution (except for a few time ranges about a year after $E2_3$). In contrast to Gogs, the cochange network of Gitea became scale-free again and stayed scale-free in most of the subsequent time ranges. The issue networks of Gogs and Gitea were scale-free in almost all time ranges. Regarding small-worldness, Gogs and Gitea show similar results: While the issue networks of both projects were small-world almost all the time (with an exception of three time ranges in Gogs almost 5 years after $E2_3$), the cochange networks were not small-world in most of the time ranges, except for a single time range that contains $E2_1$–$E2_3$ and a few time ranges some years later.

Regarding other network metrics, we observe a constant decrease in the average degree, for the cochange and for the issue network of Gogs, around 1 year after $E2_3$. Although there was a short decrease in the cochange network already shortly after this event, the degree started to grow again after two time ranges until it decreased 1 year later. The average path length of the cochange network of Gogs is largely fluctuating, beginning two ranges after $E2_3$, but was merely constant before. Moreover, the density of the cochange network decreased in the year before $E2_1$ and increased after $E2_3$ for more than a year, before it decreased again. The global clustering coefficient shows similar changes, for both the cochange and the issue network, while the network average clustering coefficient remained largely constant. Furthermore, the cochange network of Gogs started with a hybrid hierarchical structure, but the structure changed its shape more than a year after $E2_3$: The number of developers in the hierarchical part as well as in the non-hierarchical part drastically decreased, while the cochange network of

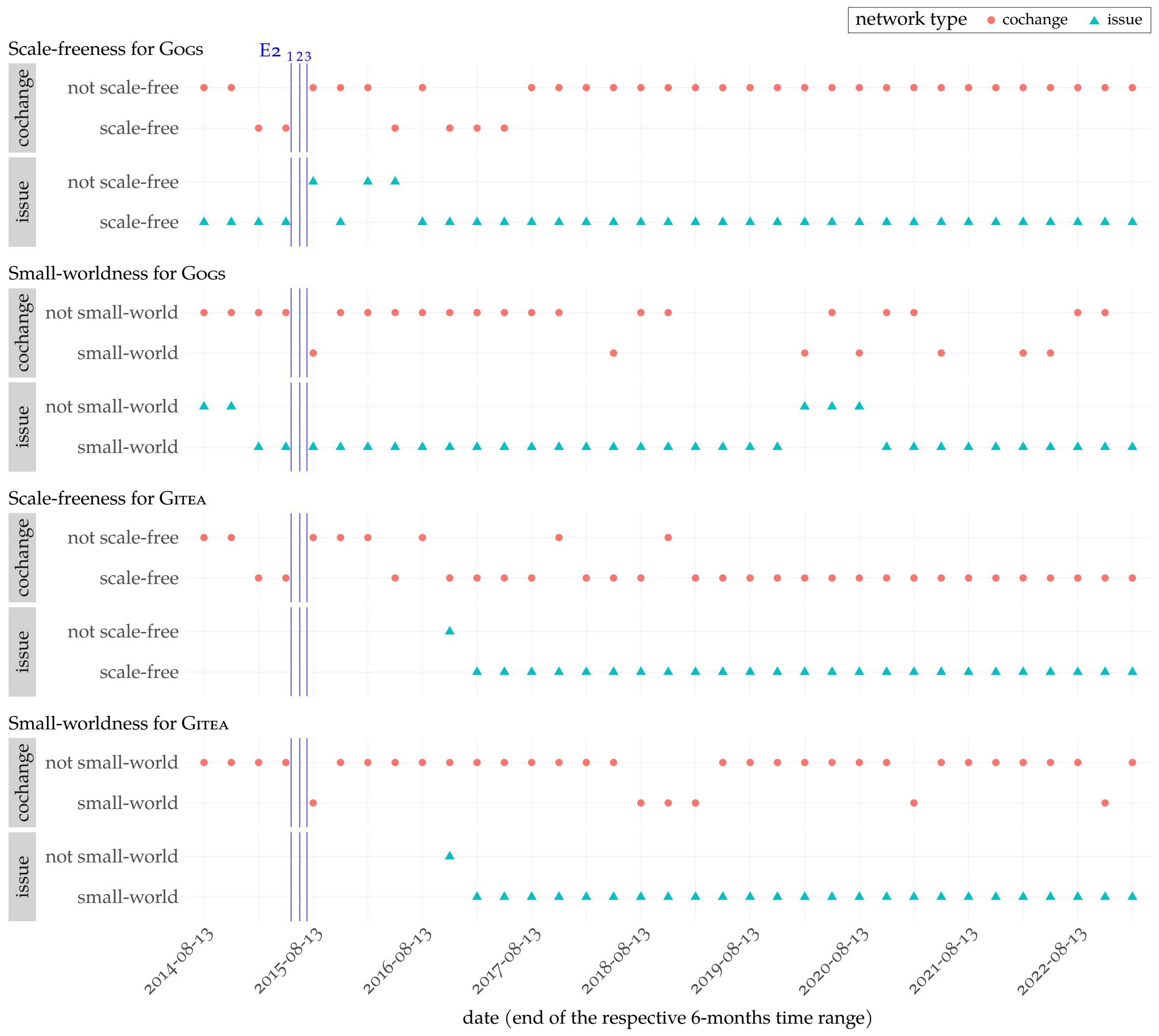


Figure 7.3: Scale-freeness and small-worldness of the cochange and issue networks for projects Gogs and Gitea over time. The blue vertical lines denote events $E2_1$–$E2_3$.

Gitea roughly continued the structure that Gogs showed until $E2_3$ and shortly thereafter. The issue network of Gogs evolved similarly to its cochange network. In contrast, the issue network of Gitea, which was created about 1 year after $E2_3$, started with the majority of developers in the hierarchical part, while the non-hierarchical part strongly grew during the first 2 years of its existence, until it reached a similar structure to what Gogs had in the year after $E2_3$.

htop The project htop in its original repository showed few activities and only few code-contributing developers, while the number of participants in the issue discussions increased and reached its maximum of almost 200 participants about 1 year before $E3_1$. Then, the number of issue participants and activities decreased until the migration of the project to another repository was finished in $E3_3$, but there was still issue activity after the cochange activity has stopped at $E3_1$. Even in the new repository, during the first year after the migration, there were a similar number of issue participants than at the maximum of the original repository, and the number of code contributors even was higher than in the original repository.

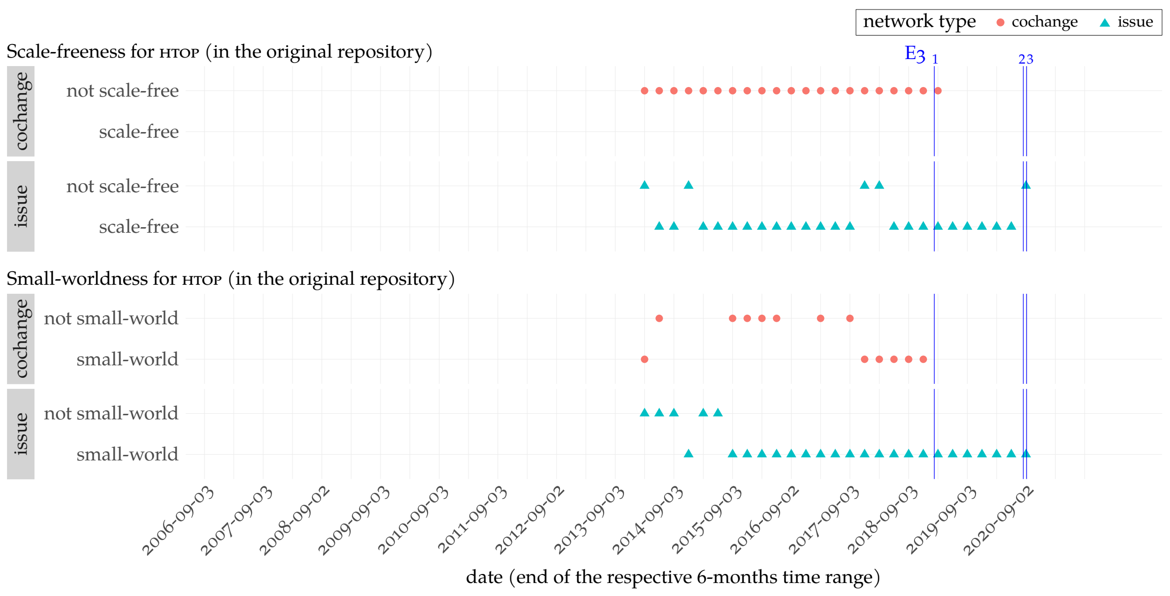


Figure 7.4: Scale-freeness and small-worldness of the cochange and issue networks for project HTOP in its original repository over time. The blue vertical lines denote events $E3_1$–$E3_3$. Missing values at the beginning correspond to a time period in which only a single developer was active. Missing values at the end correspond to the time period in which the project was continued in another repository.

In Figure 7.4, we show the scale-freeness and small-worldness characteristics of HTOP in its original repository. As there was only one developer at the beginning of the project, scale-freeness and small-worldness cannot be determined at early project stages. Later on, the cochange network was not scale-free, while the issue network was mostly scale-free (with a few outliers). Regarding small-worldness, we observe a similar picture: The issue network was almost always small-world, while the cochange network became small-world only in the last few time ranges before $E3_1$. Even in the new repository (for which we provide visualizations on our supplementary website), after $E3_3$, the cochange network was neither scale-free nor small-world, while the issue network remained scale-free and mostly small-world.

Prior to $E3_1$, the average eigenvector centrality of the cochange network was largely fluctuating between 0.2 and 1.0, and the average path length decreased from 20 (about 4 years before $E3_1$) to 2 at $E3_1$. The respective characteristic of the issue network remained largely constant, though, and decreased significantly between $E3_1$ and $E3_3$. Regarding hierarchy, we can see in the issue network that, already about 2 years before $E3_1$, the hierarchical part degenerated and mostly developers that have a high degree and a high clustering coefficient (that is, developers that are located close to the upper right corner of the hierarchy plots that we have introduced in Chapter 5, see Figure 5.3 on page 124) appeared. Although a hierarchical structure emerged again in the year before $E3_1$, it completely disappeared after $E3_1$. In the cochange network, in most of the time ranges, there were too few developers active to determine any hierarchical or hybrid structure.

Node.js In Node.js, the number of developers was very small until $E4_2$, and it slowly increased afterwards, reaching its maximum value between $E4_3$ and $E4_4$. In contrast, the number

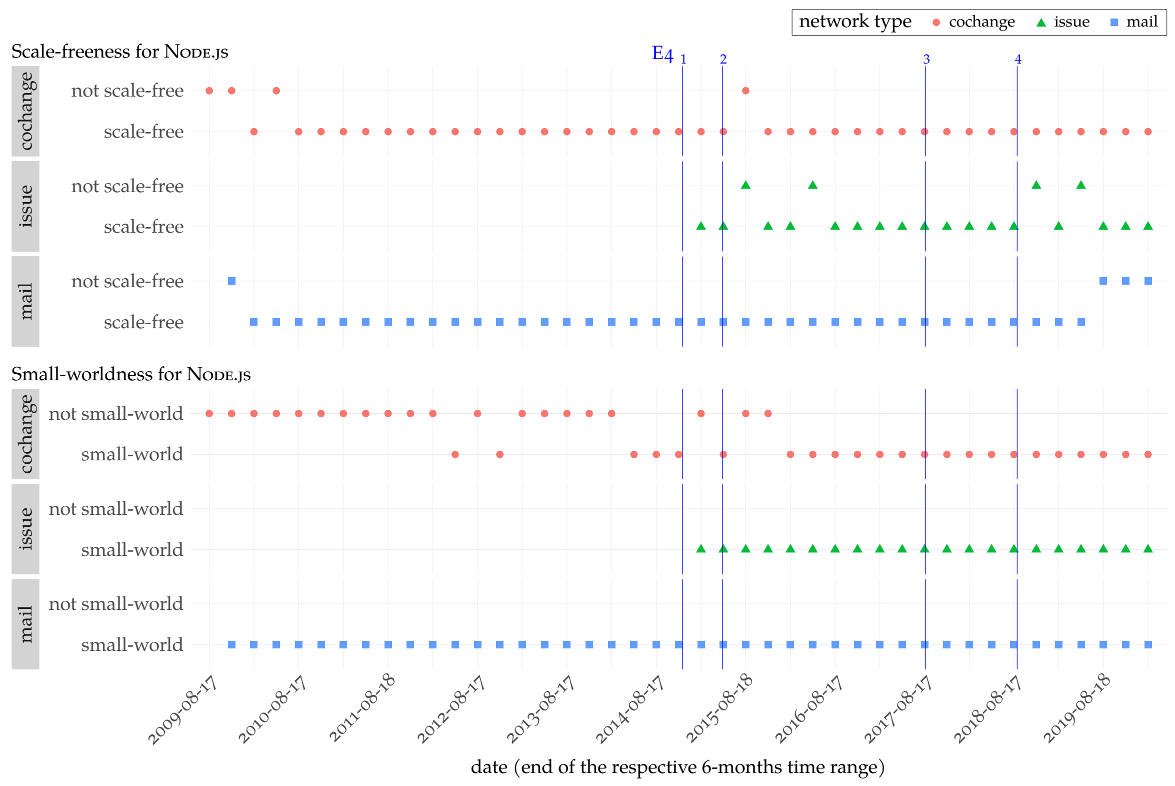


Figure 7.5: Scale-freeness and small-worldness of the cochange, issue, and mail networks for project Node.js over time. The blue vertical lines denote events $E4_1$–$E4_4$.

of mailing-list contributors was greater than 1000 at its maximum about 3 years before $E4_1$, and then it steadily decreased. The GitHub issues were started shortly after $E4_1$ and reached their maximum number of participants shortly after $E4_3$, before it also steadily decreased. $E4_1$ marks a turning point in the number of commits, which was largely constant before, but started to grow afterwards until the end of our observation period. For the number of issue comments, we observe a different picture: After the project had started to use issues after $E4_1$, we see a turning point in the middle between $E4_3$ and $E4_4$, where the number of issue comments stopped increasing and started to decrease until the end of our observation period.

The cochange network of Node.js appeared to be scale-free almost all the time (except for one range shortly after $E4_2$), as we show in Figure 7.5. However, this does not hold for small-worldness. The cochange network was not small-world at the beginning. During the year before $E4_1$, it became small-world, but then we observe four overlapping 6-months time ranges of fluctuations around $E4_2$. Afterwards, the cochange network consistently held the small-world property. In contrast, the issue network and the mail network were always small-world, and almost always scale-free (with a few exceptions in the issue network shortly after $E4_2$ and shortly after $E4_4$).

The average degree for the issue network started with its maximum directly after the beginning of the use of issues and drastically decreased after $E4_2$, but then it remained constant until the end of our observation period. This also holds for the global clustering coefficient of the issue network. The modularity of the mail network increased between $E4_1$ and $E4_3$, whereas it stayed largely constant for the issue and cochange networks. Furthermore,

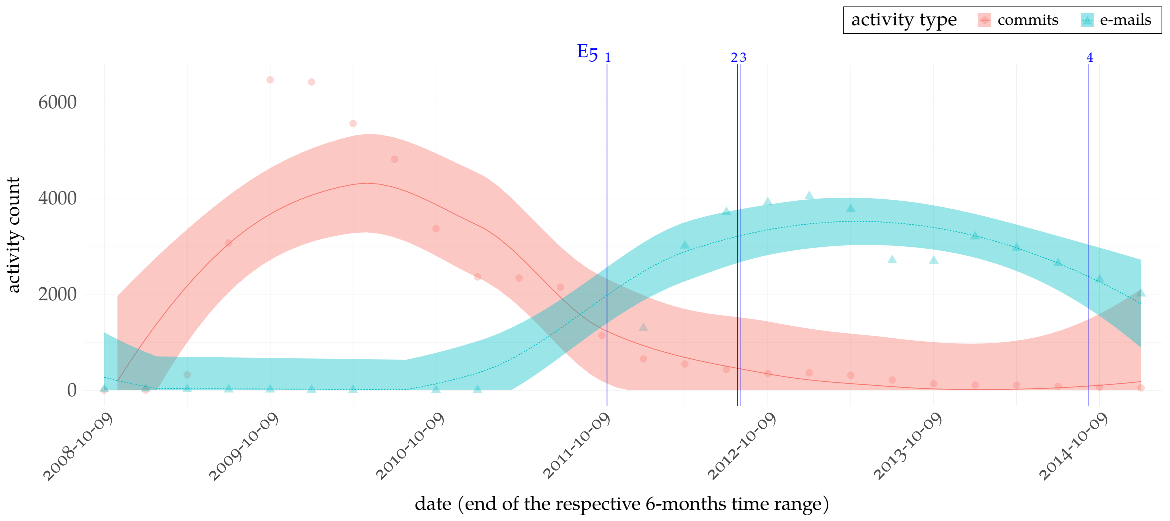


Figure 7.6: Activity in project QT: Before $E5_1$, there was mostly commit activity but almost no e-mail activity. After $E5_1$, however, it was vice versa, which is a rather uncommon phenomenon.

the hierarchical structures of the issue network and of the mail network both started with a large non-hierarchical part, which decreased over time (for the issue network after $E4_2$, for the mail network already more than a year before $E4_1$). In the mail network, the non-hierarchical part disappeared almost entirely around $E4_3$.

QT Project QT shows uncommon activity patterns (see Figure 7.6): While there were many commits around 2 years before $E5_1$, the commit count decreased since then until $E5_2$. In contrast, on the mailing list, there was almost no activity until $E5_1$, and it reached its maximum activity shortly after $E5_3$, before it decreased toward $E5_4$ and even thereafter. These opposed activity patterns are also reflected in the numbers of commit authors and e-mail authors.

The cochange network appeared to be scale-free (except for three ranges in the beginning and two ranges shortly before $E5_4$), but stopped being scale-free after $E5_4$ (see Figure 7.7). However, the small-worldness property shows some fluctuations, especially shortly before $E5_1$ and between $E5_3$ and $E5_4$. For the mail network, we can see that it was scale-free and small-world all the time in its activity period, starting at $E5_1$ until the end of our observation period.

The average degrees of cochange and mail networks follow the activity patterns of the respective data source. In contrast, the global clustering coefficient of the cochange network decreased from the beginning until $E5_2$, increased again until the middle between $E5_3$ and $E5_4$, and decreased again toward $E5_4$. While the cochange network started with a large non-hierarchical part, the non-hierarchical part almost disappeared after $E5_1$. For the mail network, we observe a different picture: Here, the non-hierarchical part increased after $E5_1$, and, later on, it stayed largely constant until the end of our observation period.

MySQL & MariaDB On the mailing list of MySQL, we can see the highest number of contributors in 2003. Afterwards, the number of contributors on the mailing list drastically decreased until $E6_1$. Then, it increased again until shortly after $E6_3$, at which point in time it started to decrease again. Independent of that, the usage of GitHub issues started after $E6_{12}$.

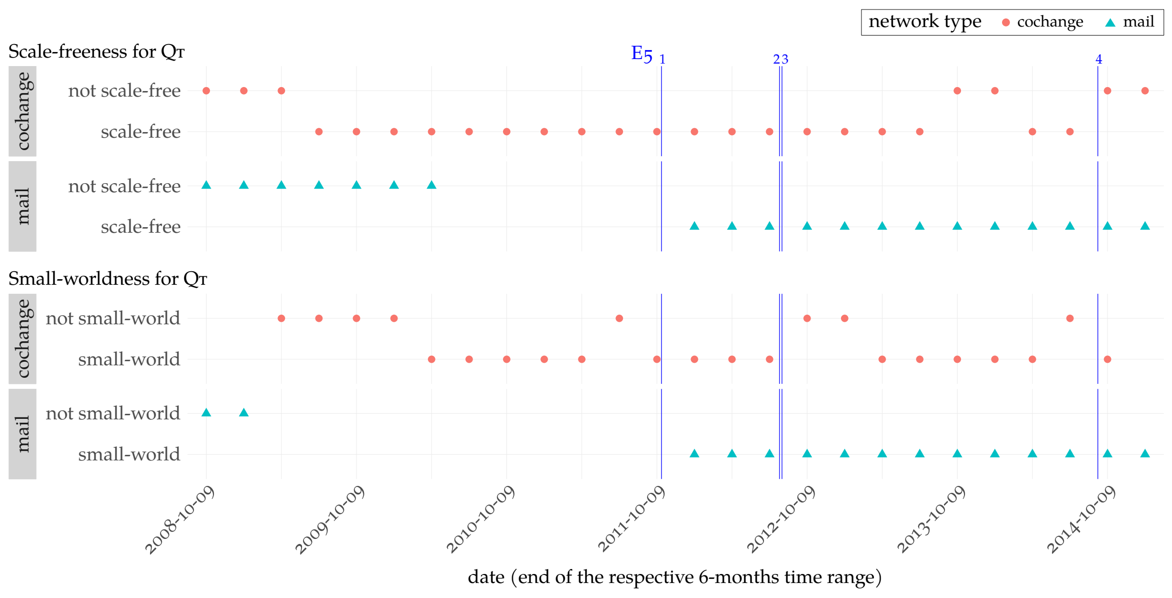


Figure 7.7: Scale-freeness and small-worldness of the cochange and mail networks for project Qt over time. The blue vertical lines denote events $E5_1$–$E5_4$.

The commit count evolved similarly to the e-mail activity: In the years before $E6_1$, we observe the highest commit count. Between $E6_1$ and $E6_8$, we observe a decreased commit count. Then, between $E6_8$ and $E6_{10}$, we again observe a higher commit count, but not as high as before $E6_1$. After $E6_{10}$, the commit count fluctuated around an average size. MariaDB shows a slightly higher commit count after its creation ($E6_6$) than before,[57] but it decreased again until $E6_{10}$. In two time ranges shortly after $E6_{10}$, we observe the highest commit count, which is about four times higher than in the ranges before or thereafter. Then, also for MariaDB, the commit count returned to an average size. After $E6_{13}$, the issue activity in MariaDB started to increase.

In Figure 7.8, we show that the cochange network of MySQL became scale-free many years before $E6_1$. At around $E6_1$, it was not scale-free, and also not in a few ranges after $E6_7$/$E6_8$. Opposed to that, the cochange network of MariaDB, shows more fluctuations: It was not scale-free in the two time ranges after $E6_3$, and, moreover, after $E6_{13}$, the presence of the scale-freeness characteristic changed very frequently. When looking at the small-worldness property, we observe that, in the cochange network of MySQL, it was present only between $E6_3$ and $E6_5$ and shortly thereafter, as well as once shortly after $E6_{10}$ and at the end of our observation period. For the cochange network of MariaDB, however, we observe small-worldness in some of the ranges after $E6_7$ until shortly after $E6_{13}$. The issue network of MySQL was pretty much never scale-free and only sometimes small-world, while the mail network was almost always scale-free (with an exception at $E6_6$) and always small-world (up to its abandonment). A few ranges after its initiation, the issue network of MariaDB became scale-free, while the small-worldness property was fluctuating from the beginning.

The average degree in the cochange network of MySQL shows high values before $E6_1$ and after $E6_{13}$, while its lowest values can be observed after $E6_3$. However, the density of the cochange network decreased from project begin until around $E6_5$, then it slightly increased

57 Note that the commit history of MariaDB differs from the history of the original project MySQL, as not all commits from MySQL became part of MariaDB, following an unusual forking process.

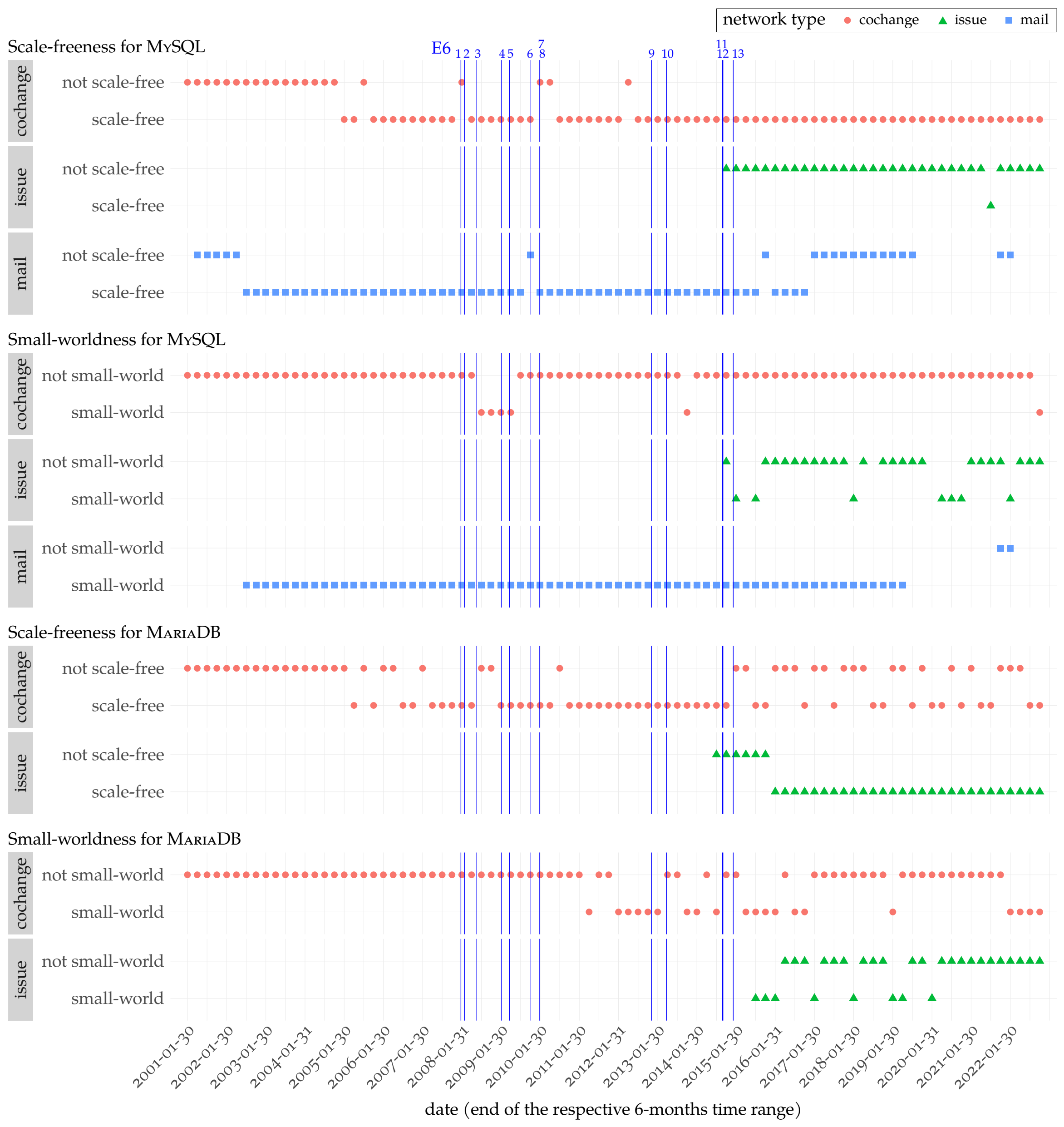


Figure 7.8: Scale-freeness and small-worldness of the cochange, issue, and mail networks for projects MySQL and MariaDB over time. The blue vertical lines denote events $E6_1$–$E6_{13}$.

and stayed rather constant. This also holds for the global clustering coefficient. Instead, the global clustering coefficient of the mail network of MySQL increased from its beginning until the end of our observation period. The hierarchical structure of the cochange network changed its shape between $E6_1$ and $E6_2$, that is, the non-hierarchical part decreased and the hierarchical part increased. This trend continued afterwards, but stopped between $E6_7$ and $E6_8$. From then on, the non-hierarchical part tended to become greater again. In MariaDB, we see similar results for the cochange network. However, while the decrease in density and global clustering coefficient already ended at about $E6_5$ for MySQL, the decrease for MariaDB ended later at about $E6_{10}$. Interestingly, the modularity of the cochange network of MariaDB slightly increased from 0 to about 0.18 between $E6_8$ and $E6_9$. Then, it decreased again (with a few outliers after $E6_{10}$). In contrast to MySQL, MariaDB had a relatively sparse hierarchical structure (only for very high degrees, there is a hierarchical structure detectable).

OpenOffice & LibreOffice & Apache OpenOffice While LibreOffice kept the commit history of OpenOffice, Apache OpenOffice imported the complete history of OpenOffice in a single commit at its creation. For LibreOffice, the commit activity was largely fluctuating throughout its entire life time, as already in the history of OpenOffice before the creation of LibreOffice. A few months before $E7_1$, we observe the highest commit count, and around $E7_1$, we observe the lowest commit count. Afterwards, we can see some local maxima before $E7_4$ and before $E7_8$. The e-mail activity reached its maximum around $E7_7$ and constantly decreased afterwards. For Apache OpenOffice, we observe a high commit activity around its initiation, then the activity decreased between $E7_7$ and $E7_9$, and eventually it remained constant at a low level. For the e-mail activity of Apache OpenOffice, we observe a similar picture as for the commit activity. Issues play a subordinate role in both LibreOffice and Apache OpenOffice.

In Figure 7.9, we can see that the cochange network of LibreOffice was not scale-free until shortly before $E7_2$. From then on, it was scale-free (with one exception between $E7_6$ and $E7_7$). In contrast, the cochange network of Apache OpenOffice was never scale-free. However, the mail network was mostly scale-free for both LibreOffice and Apache OpenOffice (with an exception in LibreOffice shortly before $E7_{11}$, and a few exceptions in Apache OpenOffice, notably directly after $E7_{10}$). Only in some rare cases, the cochange network of LibreOffice was small-world, especially between $E7_3$ and $E7_7$, and also before $E7_{11}$. For the cochange network of Apache OpenOffice, we cannot draw any conclusions, as there was only low activity in the project and, thus, small-worldness cannot be determined in most of the ranges. Only around $E7_9$, we observe three small-world ranges, and, from then on, there occasionally are ranges in which the cochange network was small-world. Due to low activity, the small-worldness of the issue networks of LibreOffice and Apache OpenOffice cannot be determined in most of the ranges. For mail networks, we can see that the mail network of LibreOffice was always small-world, and that the mail network of Apache OpenOffice was almost always small-world.

While OpenOffice started with a comparably high average degree in the cochange network, it decreased toward $E7_1$. Then, in LibreOffice, we observe phases of high average degree between $E7_3$ and $E7_4$, and also between $E7_7$ and $E7_8$. Even after $E7_8$, the average degree remained at a comparably high level. Regarding hierarchy, we observe the increase of the non-hierarchical part of the cochange network at about $E7_1$ or $E7_2$. In the mail network, we can observe the opposite behavior: At fork creation, there was a large non-hierarchical part, but after $E7_7$, the network tended to consist mostly of a hierarchical part. In Apache OpenOffice, the average degree of the cochange network started with a comparably high value and decreased between $E7_7$ and $E7_9$. However, on the mail network, the average degree increased between $E7_8$ and $E7_{11}$. While global clustering coefficient and network average clustering coefficient of the mail network of Apache OpenOffice largely stayed constant (with a slight increase over time), the clustering coefficients of the cochange network strongly fluctuated between 0 and 1. Notably, there were too few developers active in Apache OpenOffice to be able to determine a hierarchical structure in the cochange network. However, in the mail network, there is a clearly visible hybrid hierarchical structure.

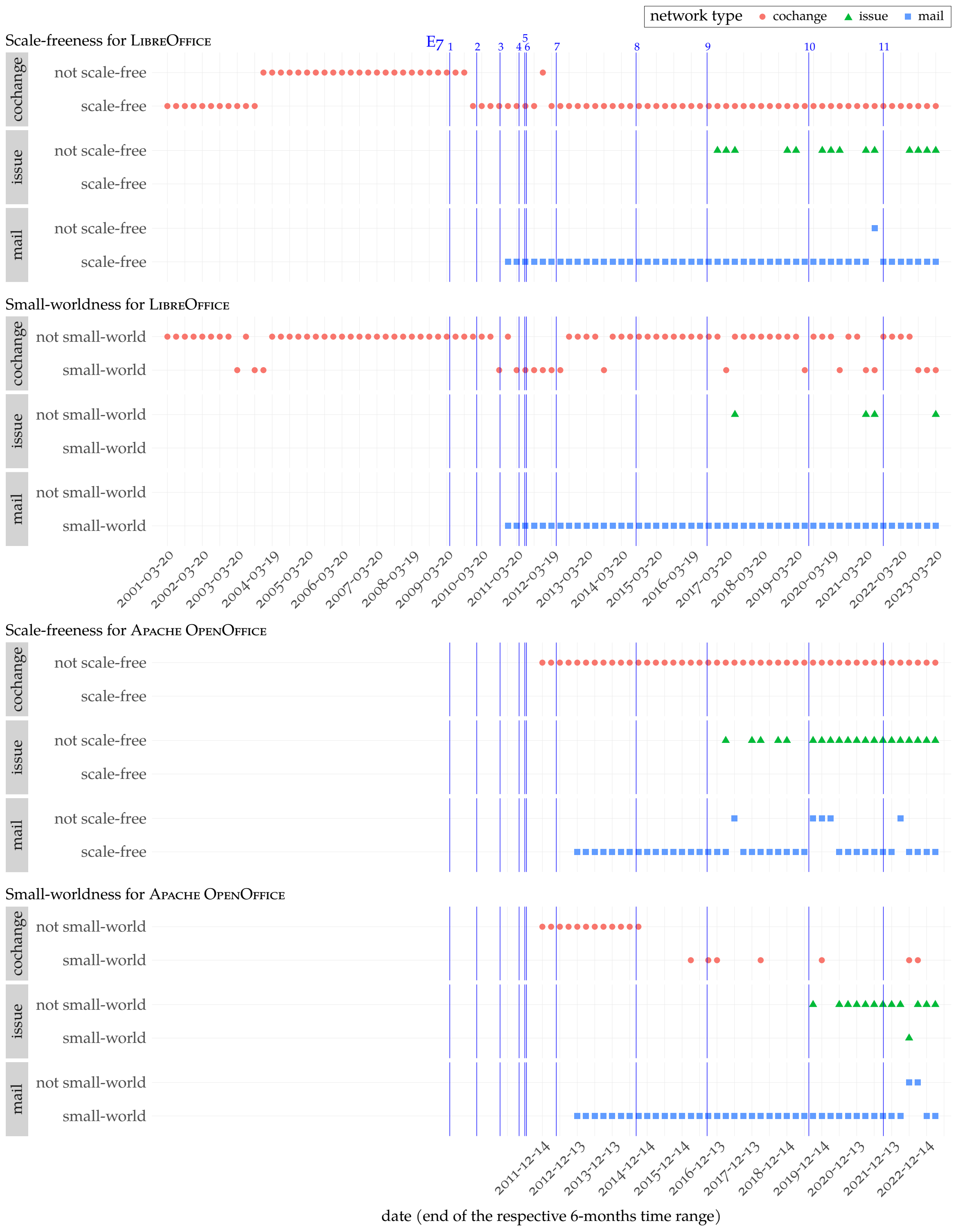


Figure 7.9: Scale-freeness and small-worldness of the cochange, issue, and mail networks for projects LibreOffice and Apache OpenOffice over time. The blue vertical lines denote events $E7_1$–$E7_{11}$. While LibreOffice forked the source-code repository from OpenOffice at $E7_3$ and, thus, kept the commit history of OpenOffice, Apache OpenOffice initially imported the complete source code of OpenOffice in a single commit shortly after $E7_5$ and, thus, lost the commit history of OpenOffice.

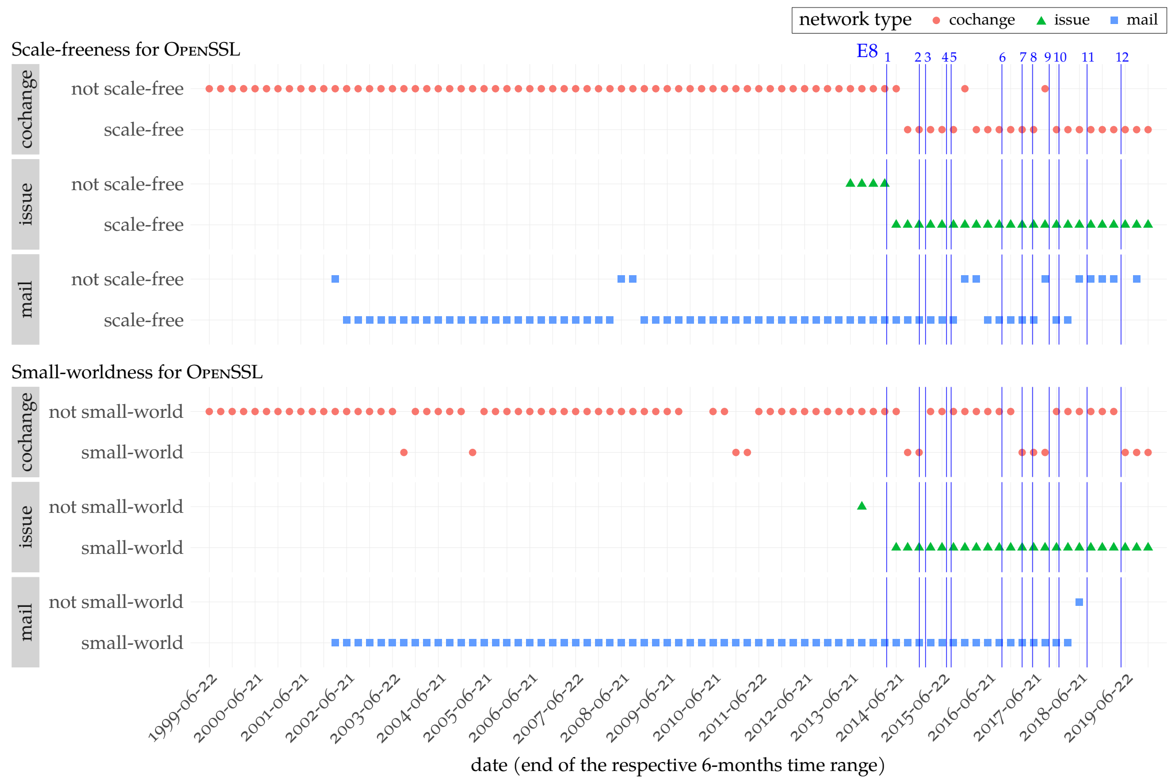


Figure 7.10: Scale-freeness and small-worldness of the cochange, issue, and mail networks for project OpenSSL over time. The blue vertical lines denote events $E8_1$–$E8_{12}$.

OpenSSL After $E8_1$, we observe a strong increase in the commit activity of OpenSSL, which continued until shortly before $E8_6$. Then, the commit activity decreased until $E8_{12}$. For the activity on the mailing list, we see an increased e-mail count between $E8_1$ and $E8_3$ as well as between $E8_5$ and $E8_6$. Afterwards, the e-mail count decreased and reached a value of 0 after $E8_{11}$. In contrast, from $E8_4$ on, the issue activity increased until the end of our observation period.

In Figure 7.10, we show that the cochange network of OpenSSL was not scale-free until shortly after $E8_1$. For the remaining time, it was scale-free (except for one range after $E8_5$ and another range shortly before $E8_9$). The cochange network was not small-world for almost all the time. Nevertheless, there are a few exceptions shortly before $E8_2$, between $E8_6$ and $E8_9$, and after $E8_{12}$, in which the cochange network was small-world. In contrast, the mail network was small-world all the time, and it was also scale-free for most of the time. The ranges between $E8_5$ and $E8_6$ and shortly before $E8_9$, in which the mail network was not scale-free, coincide with the time ranges in which the cochange network was not scale-free. Remarkably, from $E8_1$ on, the issue network was small-world and scale-free without any exception.

The average degree in the cochange network reached its maximum value between $E8_5$ and $E8_6$, while the average degree in the mail network as well as in the issue network reached its maximum shortly after $E8_{10}$. While the average eigenvector centrality in the cochange network was about 0.5 before $E8_1$, it went down to about 0.1 already a year before $E8_1$ and continued decreasing until $E8_2$. This also holds for the density in the cochange network, which, however, slightly increased after $E8_2$. A hierarchical structure in the cochange network started to establish around 1.5 years before $E8_1$. After $E8_1$, the non-hierarchical part started to increase slightly over the next years. In contrast, the mail network had a hierarchical structure from the beginning on, but the residual variance of the corresponding regression fit was comparably large.[58] After $E8_1$, the variance decreased, but also the non-hierarchical part started to increase. Also, the issue network shows an hierarchical structure from the beginning on.

7.3.3 Discussion of the Observations

Above, we have presented our observations regarding network characteristics, for each case study. Now, we briefly discuss them with a focus on the affected organizational events.

ownCloud & Nextcloud We observe changes in the number of developers and their activities in project ownCloud already during the 2 years before the resignment of the founder of ownCloud and the start of Nextcloud. These changes also affected the network characteristics of the cochange network, as density, average degree, average path length, and global clustering coefficient change when the number of activities change. For small-worldness and scale-freeness, we observe changes only about a year after the investigated events. These changes clearly show that ownCloud and Nextcloud have diverged, also in their network structure: ownCloud lacks small-worldness and scale-freeness, while Nextcloud achieves these properties. Regarding small-worldness, it is also remarkable that the cochange network is small-world exactly at the point in time at which the fork was created.

Gogs & Gitea Although the creation of Gitea as a fork of Gogs displays a similar kind of event as the creation of Nextcloud as a fork of ownCloud, the circumstances are different: While there was already a breakdown in developer activity before the fork creation in ownCloud, this phenomenon does not apply to Gogs, since its fork Gitea resulted from a temporary unavailability of the main developer of Gogs.[10] It also took more than a year after fork creation until Gitea has established its own issue tracker, and also in the networks' various characteristics, we observe changes only at that point in time, underlining that the actual spin-off between fork and original project took place 1 year later than the actual announcement and start of the fork had happened. Shortly after the creation, but also in the long run, the cochange network of Gitea achieves scale-freeness, and in some rare ranges, even small-worldness, while the original project Gogs does not achieve these properties.

htop Project htop mainly relied on a single developer, who was inactive from time to time, especially in the year before he stopped contributing. Therefore, the activity in the project already decreased during this year, which also affected the network characteristics of the cochange network (e.g., large fluctuations in average path length), while the communication activity was still going on and only decreased when the migration to a new repository started. Noteworthy, the hierarchical structure of the issue network already started to change during the years before the migration, which also might be caused by the temporary periods of inactivity of the founder of htop. Also worth to note is that the cochange network became small-world in the year prior to the last activity of the founder, which needs further investigation on how and why this has happened.

58 See Figure 5.3 on page 124 for an explanation of what we mean by variance and regression fit here.

Node.js For Node.js, we see the creations of its forks io.js and Ayo.js at different points in time as drastic events that affect the developer activity differently for commits and communication: While the commit activity is mostly unaffected (it even increases after the creation of the forks, but this may have happened due to other reasons), the issue communication was initiated after the first forking event and reached its maximum activity around the second forking event. A reason for this could be that the potential behavioral violations, which caused the forks, may have created a need for clarification and discussions about project organization among the developers. Hence, it is also necessary to highlight that the forks were caused by specific incidents in developer behavior, and that both forks only stayed active for not longer than 1 year. While we observe some fluctuations on the small-wordness of the cochange network around the time of the creation of the io.js fork, the network started to be small-world already some time before io.js, which is why we assume that the change in the small-worldness might have other reasons than just the emergence of the fork.

Qt The switch to an open-governance model in Qt affected that there suddenly was high activity on its mailing list, but the commit activity continued to vanish. We hypothesize that the switch of the governance model attracted new developers who started to discuss their needs and how they would like to adjust the software product. However, we assume that they merely laid out a plan for how to achieve these changes instead of implementing them directly. This is in line with the fact that project version Qt4 was discontinued about 3 years after the switch and that its successor Qt5 was initiated using a completely new implementation architecture in a new repository.[59] Also the department closure in Brisbane and the acquisitions of Qt by different companies may have led to the vanishing commit activity and to the temporary loss of the small-worldness property of the cochange network, contributing toward the new implementation architecture in Qt5.

MySQL & MariaDB MySQL had its highest project activity before it was taken over by Sun. However, after the completion of its acquisition, there was also a temporary increase in developer participation until the VCS was changed. Another temporary increase in commit activity happened after the acquisition by Oracle and during the time at which a previous core developer left the project. MariaDB started with a higher activity count than MySQL had directly before the fork creation. Also, after SkySQL was merged into MariaDB we observe a temporarily increased commit activity.

The cochange network of MySQL was almost always scale-free, but not during the time in which Sun performed the acquisition. During this time, also the hierarchical structure of the project changed. Another change in the hierarchical structure happened when Oracle performed the acquisition of Sun, which also led to a change in density and global clustering coefficient. Also, after a longstanding core developer had left MySQL, the project lost its scale-freeness for one time range, until the network structure has normalized again. The small-worldness property was only present after switching VCS technologies, but only for a short time until MariaDB was created. On the other hand, the technology switch also led to a short-term decrease in average degree.

59 Lars Knoll: "Thoughts about Qt 5", 2011: https://www.qt.io/blog/2011/05/09/thoughts-about-qt-5/ (accessed at 2023-11-30)

The cochange network of MariaDB, however, shows different fluctuations than MySQL and shows spurious patterns. On the one hand, this might be caused by the fact that the commit history of MariaDB misses several commits that are contained in the history of MySQL. On the other hand, changes in the company structures behind MariaDB might be related to the fluctuations. Remarkably, after MySQL has migrated its VCS to Git, the scale-freeness of MariaDB was largely disturbed. While this might be just coincidence, this could also mean that some MariaDB developers contributed to MySQL and temporarily decreased their activities in MariaDB, as now both projects used Git. In general, the cochange network of MariaDB seems to have a less pronounced hierarchical structure than the cochange network of MySQL.

OpenOffice & LibreOffice & Apache OpenOffice In general, the participation in OpenOffice was highly fluctuating. We notice an activity decrease around the announcement that Oracle will buy Sun, and an increased commit activity before the announcement of the stop of OpenOffice development and also before the withdrawal of IBM from Apache OpenOffice. We hypothesize that these announcements involve both, leaving developers who worked on OpenOffice on behalf of the involved companies, and volunteer developers who stopped contributing as they either expected the project to be discontinued or unmaintained or as they disagreed with future development plans of the project.

The cochange network of OpenOffice became scale-free during the time period in which Oracle performed the acquisition of Sun. Thus, the transition to the new company may have involved a major change in the developer interaction. Indeed, during this time, also the hierarchical structure of cochange and mail networks changed. Another major change to the network structure happened after the creation of LibreOffice, as in this fork the cochange network started to be small-world.[60] However, this changed after the start of Apache OpenOffice—maybe some developers moved from LibreOffice to Apache OpenOffice. There are also some other fluctuations in the small-worldness in later stages of LibreOffice, but as they happened prior to unforeseeable events (e.g., security vulnerability disclosure) or are related to internal structures of Apache OpenOffice, we do not relate these fluctuations to these events.

As the number of contributors for Apache OpenOffice was comparably low (see Table 7.3), the corresponding networks are too small (in terms of the number of vertices) to draw any conclusions about changes in their network structure. Nevertheless, the average degree of its cochange network decreases between the creation of Apache OpenOffice and the discussion about potential retirement, which confirms the low activity and the necessity of such discussions in the project. Another interesting phenomenon is that the mail network of Apache OpenOffice lost its scale-freeness property shortly after the migration from SVN to Git, which might also have involved a change in the communication processes.

OpenSSL The disclosure of the Heartbleed vulnerability led to a strong increase in developer activity, as already indicated by previous work [Wal20]. After code-cleanup and technical migrations had been finished 2.5 years later, which had been started as a consequence of Heartbleed, the developer activity decreased again. Later license changes or changes in the management committee did not affect the activity in the project.

60 Note that we analyzed the source-code repository of LibreOffice, which contains the commit history of OpenOffice before the creation of LibreOffice. To the best of our knowledge, the original source-code repository of OpenOffice is not publicly available any more.

While the cochange network of OpenSSL was not scale-free when Heartbleed was detected, it became scale-free roughly half a year thereafter, as first measures of organizational change had already been applied to OpenSSL. Small-worldness, however, was only achieved temporarily during the large changes of the code base, as well as later on after the license change when the management committee was established and two co-founders left the project. Noteworthy, about 1.5 years before the disclosure of Heartbleed, several network characteristics (e.g., density, average eigenvector centrality, etc.) decreased and a hierarchical structure started to establish. After Heartbleed, especially the non-hierarchical part becomes more prevalent and important in the network, which we attribute to the publicity of Heartbleed and its severity. Since many other projects and tools were drastically affected by this vulnerability, we assume that many outsiders started to join OpenSSL to make sure that they can close the security gaps in their products and to prevent future vulnerabilities of this severity in OpenSSL.

7.4 Perspectives

Based on the observations from our exploratory study, we summarize our results, derive hypotheses based on these results, and provide perspectives on how our research can be continued and practically used.

Answer to our RQ: As we have presented above in our exploratory study on 8 case studies, we have seen that there are particular network characteristics that reflect changes in the network structure around the points in time when specific organizational events happened. We observed such changes mainly in the cochange networks, rather than in the communication networks. However, we were not able to detect such changes in the network structure for each organizational event, but only for a subset of the investigated events. Particularly, it is important to note that our observations are of a preliminary nature and are very project specific. Therefore, further investigations are necessary to confirm our observations qualitatively and quantitatively in a more general setting.

As already indicated above, most of the changes in particular network characteristics that we observed affected the cochange networks. While this could just be due to coincidence, we also come up with potential reasons for that: The cochange network contains all the developers who contributed to the source code in a specific time range. Thus, the developers who are part of a cochange network are the developers who are responsible for the entire source-code implementation, which is the key part of the resulting software product. Therefore, essential changes that affect the project's success, of course, should affect the cochange network. In contrast, communication channels—may it be a mailing list or an issue tracker—also contain users and other developers who do not necessarily contribute to the source code but primarily discuss the source code and who behave differently than actual developers who actively contribute to the project's source code [BLJ+13; DOB+18; GBL+13; MAJ+20; OHM+18; TDH14]. Therefore, the number of vertices in the communication networks usually is much larger than in the cochange network, as can be seen in Table 7.3. Consequently, structural changes in the interaction of developers may merely affect the cochange network, as it is smaller than the communication network and, thus, more susceptible to change. Moreover, the cochange network is primarily composed of the core contributors, which play only a minor role in the communication networks by the sheer number of people involved in the discussions. Therefore, we suggest that also future work on analyzing network characteristics with respect to organizational events should focus on cochange networks.

> **Hypothesis 1:** Developer networks that reflect cochange interactions between developers are more susceptible to structural changes that are related to organizational events than developer networks that reflect communication among developers.

While our exploratory study revealed that many of the investigated network characteristics change temporally close-by to organizational events, some of these characteristics simply depend on the number of developers and their number of activities. For example, the average degree, the average path length, or the density depend on the participants and activities in a specific time range. That is, an increase in the commit count or in the developer count usually leads to an increase in degree and density and to a decrease in average path length. As this sounds rather obvious, and since these characteristics do not add additional value compared to pure count-based characteristics, future work should focus on investigating rather complex network characteristics that combine multiple different aspects into one characteristic, such as scale-freeness, small-worldness, or hierarchy.

Our exploratory study showed that scale-freeness and small-worldness are two properties that regularly are subject to fluctuations. However, there are also large time periods of consistency in these properties. Thus, we were able to notice that changes in scale-freeness and small-worldness often go hand in hand with specific organizational events. Especially, events that are related to strong developer turnover [FPB+15; FSV20; IRO+09; JYF24; Moc10] or developer attraction [FHV23; SGR14] (e.g., acquisitions by companies, creations of forks, disclosure of severe vulnerabilities, or switches of technologies) often lead to temporary or sometimes even long-term changes in scale-freeness or small-worldness. In particular, we observed situations in which projects gained these properties temporally close-by to such events, but we also observed situations in which projects lost these properties temporally close-by to such events. Albeit, the lack of small-worldness or scale-freeness potentially could have negative effects on the coordination and information flow among developers in a software project, as both properties are assumed to be beneficial for successful project coordination [JAM17; Sin10].

> **Hypothesis 2:** A change in the scale-freeness property of cochange developer networks is related to specific organizational events that affect the structure of the network.
>
> **Hypothesis 3:** A change in the small-worldness property of cochange developer networks is related to specific organizational events that affect the structure of the network.
>
> **Hypothesis 4:** The hybrid hierarchical structure of developer networks, consisting of a hierarchical and a non-hierarchical part, changes as a consequence of or in anticipation of the occurrence of specific organizational events.

Another aspect that is worth to be discussed is the temporal dimension of organizational events. While, for some of the investigated events, we observed changes in network characteristics directly after the event has happened or during the subsequent year, for other events, we observed changes in the network structure in the years or months before the event has happened. Of course, we cannot reliably relate these changes in the network structure to a specific event at a specific point in time—it can also be that multiple events are related to specific changes, and also unconsidered or unknown organizational events could have played an important role. Nonetheless, we suggest that future research should consider organizational events from two different points of view: On the one hand, organizational events can cause changes in the network structure as a consequence of the event happening. On the other hand, changes in organizational structure of a project (including changes to network characteristics) can cause particular organizational events. Obviously, an organizational event can also fulfill both views: It can be caused by structural changes and, at the same time, it can cause additional structural changes that occur later on. Albeit, in general, it is not trivial to reliably detect such cause-effect chains in OSS projects due to the sheer number of structural changes and the number of potential (known and unknown) organizational events. While we lay the foundations for such investigations in our exploratory study, it is necessary to complement them by future studies that analyze the effect of organizational events more qualitatively.

Hypothesis 5: Organizational events can be grouped into different categories based on their temporal dimension: Some events show only short-term effects, while others affect the project community in the long run. In addition, some events happen as a consequence of structural changes in the project community, while other events imply structural changes in the developer network of the project.

Investigating the proposed hypotheses in future studies shall contribute toward a better understanding of the organizational processes that drive and influence the evolution of the developer network, and, thus, the evolution and sustainability of the project. In perspective, our findings shall serve as foundations for the development of early-warning alarms or even countermeasures of negative organizational trends that could affect the success of a project, as indicated by the literature [e.g., CMP07; CV17; JA22; YZF23].

The observations from our exploratory study have practical value for both researchers and practitioners: For researchers, we provide insights into network evolution and organizational changes in OSS projects, which could be used to better understand contribution dynamics and organizational events, not only with respect to early-warning mechanisms but also to obtain an understanding of the cause and effects of different organizational events. For practitioners, our results provide insights into the evolution of established projects and the organizational structure behind, which could be used to avoid specific events that are negatively connotated. Moreover, these insights can be used to get to know how specific network characteristics have potentially been affected by organizational events and to look back on past events and reappraise them from a network-theoretic point of view. This way, in perspective, one can learn from past events and draw conclusions for the handling of potential events that might happen in the future and might threaten the success of a software project.

7.5 Threats to Validity

The validity of the results of our exploratory, empirical study may be threatened in various ways. Below, we discuss the potential threats by grouping them, as suggested by Wohlin et al. [WRH+12], into four categories that were introduced by Cook and Campbell [CC79].

Internal Validity The validity of our results highly depends on the case studies and corresponding events that we have selected for our study. Due to the exploratory and preliminary nature of our study, we selected projects for which the corresponding organizational events are publicly known and were discussed in the media, specific to technical topics. We admit that this could induce a selection bias for our events, but we tried to base our decisions on similar events that have been investigated in the software-engineering literature (see Sections 7.1.1 and 7.1.3). While the different kinds of events and projects that we analyze contribute toward generalizability, these differences may also affect the internal validity as the projects and events are too different to be compared with each other. However, the choice to investigate a diverse set of projects and events was made on purpose, to exploratively obtain insights into a wide variety of projects and events, as a foundation for future studies in this context.

Also, the selection of interaction channels that we investigate may affect our results. Therefore, we selected multiple interaction channels for each project, namely the collaboration on the source code, as well as communication among developers. While some projects primarily use mailing lists, others primarily use issues [EGW22; KGB+18; SSF+17]. Therefore, we selected the communication channels that were publicly available and are extensively used. As we already have seen in our results, multiple projects used mailing lists and issues—some of them use both channels in parallel, while others fade-out the mailing list in favor of issues, and still others started to use issues but still perform the majority of their communication activities on the mailing list. As we were not aware of the amount and kind of activity that takes place on either communication channel, we investigated both of them, if publicly available, to be able to find interesting activity patterns in either of the channels. This way, we base our exploratory study on a broad data basis. Still, we might omit activities that take place on private communication channels (e.g., personal e-mails) or on other communication channels that are not in the scope of our work (e.g., chat systems) [EGW22; KGB+18; SSF+17], but we selected very prominent and highly used communication channels for each project, as the descriptive statistics in Table 7.3 indicate.

Another threat to internal validity might be the way how we preprocess the data. As already discussed in previous chapters, we identify developers by matching their names or e-mail addresses. Thus, we rely on the disambiguation heuristic of Oliva et al. [OSO+12], which is reliable according to Wiese et al. [WSS+16]. Different to our previous studies (see Chapters 5 and 6), we did not filter bots in this exploratory study. This decision was made on purpose, as bots might affect developer behavior with respect to specific organizational events. Thus, removing bots could potentially hide changes in network characteristics that emerged due to interactions with bots. However, as highly active bots might distort the network structure, it might be interesting to repeat our study while removing bots in future work, to approach our research question from a different point of view. Nonetheless, as we rely on GitHub issue data for bot detection and since the detected bots usually do not contribute to the source code, this would only affect projects that use GitHub issues. Even in such projects, removing bots would primarily affect the issue networks and, thus, should not distort our observations, which are mainly based on the cochange networks. Finally, as in the previous chapters, when we analyze the commit data, we only consider implementation files, neglecting documentation files, for instance. While this might affect our results, in this thesis, we focus on the implementation-related files. Also, the number of documentation files is much smaller than the number of files that contain source code.

Construct Validity The way we construct our networks may affect the validity of our results. In particular, we decided to construct networks for time ranges of 6 months. While the results are different when using other time ranges, we settled for 6-months ranges, for multiple reasons. On the one hand, time ranges of 6 months are quite established in empirical studies in this context [see Chapters 5 and 6; HKC+11; MW11; PBD+14]. On the other hand, we performed a sensitivity analysis and also tried time ranges of 3, 9, and 12 months, for which we also provide our results for all case studies on our supplementary website. However, 3-months ranges appeared to be too unstable and too noisy with respect to short-term fluctuations, and 9- and 12-months ranges often were too long to reveal changes in the network structure that might only be identifiable within a comparably short period of time, temporally close-by to the investigated events.

It might also be that some of the selected network characteristics depend on each other (e.g., small-worldness depends on the clustering coefficient). Albeit, this does not threaten the results of our exploratory study, in which we aim at looking at developer networks from different points of view. Using different characteristics helps identify and understand specific changes in these characteristics and relating them to specific organizational events.

Furthermore, the dates that we determined for an event and the identification of events per se could affect our study. Here, we relied on blog posts and other articles that are available via web search. However, this does not mean that an event happened at exactly the point in time at which the article was published or at the date that was mentioned in the article. To mitigate this threat, we did not arrange our time ranges with respect to the dates of the events, but created them throughout the entire project history, in line with the exploratory nature of our study. Instead of looking at the exact date of an event, we looked for changes in network characteristics that appeared also in the time before or after the date that was assigned to the event.

Conclusion Validity As we deliberately did not use any statistical methods to evaluate our results, our results and conclusions might be inappropriate to some extent, as they might be based on our subjective view on the selected projects and their evolution history. However, we left over a statistical evaluation to future studies, in favor of our preliminary and exploratory way of detecting changes in network characteristics. This way, we were able to derive hypotheses that can and need to be evaluated in future studies.

Moreover, the observed changes and fluctuations in specific network metrics might not be caused by the investigated events. Thus, there also might be other reasons or other events that we did not consider that could cause these changes. Nonetheless, it was not our goal to directly relate changes in network characteristics to certain events or vice versa, but to exploratively check if there are observable changes that potentially could be related to organizational events, to be able to investigate such changes and events in more detail in future research.

External Validity While we try to draw conclusions from all of our 8 case studies and find common characteristics with regard to organizational events, it is important to note that even similar events can have different consequences in different projects. Consequently, our results do not represent any universal characteristics of OSS projects, but provide potential starting points for future investigations in that direction.

To mitigate generalizability issues, we investigated a diverse set of 53 events, 12 long-lasting and highly active projects of various project sizes and different project ages (see Tables 7.2 and 7.3), and up to 3 different interaction channels. Although our empirical study has a preliminary and exploratory character, we carried out 8 individual case studies to collect insights from many different projects.

7.6 Conclusion

The success and sustainability of an OSS project not only depends on its implementation and maintenance, but also can be influenced by external factors, such as organizational events.

In an exploratory study on 8 case studies, for which we investigated a total number of 53 events (including different kinds of events, such as company involvements, fork creations, or technology switches), we obtained various valuable insights into the relationship between the characteristics of developer networks and the occurrence of such organizational events. First, we showed the evolution of different network characteristics for different kinds of developer networks and analyzed noticeable changes. Most of these changes were found in the collaboration network that is based on source-code changes, although also communication networks undergo recognizable changes. Second, we demonstrated that we are able to observe changes in particular network characteristics that are temporally close-by to selected organizational events. Although we could not detect a consistent set of changes in network characteristics that apply to all organizational events, we examined a diverse set of different changes in combination with a diverse set of organizational events that sometimes occur prior to an event and sometimes after an event, notwithstanding that we cannot establish an actual connection between event and change in the network characteristic. Third, we collected a set of hypotheses regarding which network characteristics might be promising indicators for detecting organizational events that could affect the structure and, thus, the success of a project. Our exploratory study revealed that, especially, complex network characteristics that combine multiple aspects, such as scale-freeness, small-worldness, or hierarchy, show interesting phenomena with respect to the evolution of an OSS project.

With our exploratory analyses and detailed discussions, we lay the foundation for future work on the analysis of socio-technical aspects of OSS projects, as the concrete roles of particular organizational events in OSS projects still remain unexplored. Furthermore, we open the door for future investigations on the relationship between organizational events and project success, which can contribute toward a better understanding of external factors in OSS projects as well as toward the development of early-warning mechanisms to avoid or mitigate the consequences of negative organizational events.

8 Concluding Remarks

Many widely-used software projects are organized and developed as open-source software (OSS) projects. Due to their publicly available source code, OSS projects often attract a high number of contributors, also including volunteers [RSA+19; Rie19]. While this is a major advantage of OSS development, previous research has indicated that problems in the organizational structure of OSS projects and a lack of coordination can cause software bugs and can also threaten the sustainability of an OSS project [CMP07; CV17; EP91; HKA+23; ZCC+19]. Consequently, understanding the organizational structure of OSS projects and how it evolves over time is pivotal for improving the coordination of developers in OSS projects.

In this thesis, we have conducted five different empirical studies with respect to the emerging organizational patterns in evolving OSS projects. Our studies consist of a mixture of quantitative and qualitative methods to analyze developer activity and coordination from different points of view with the goal to obtain a better understanding of the underlying organizational processes in OSS projects. In the long run, our investigations and corresponding results shall contribute toward an improvement of organizational and coordinative socio-technical processes in OSS projects.

In what follows, we briefly summarize our contributions. Concluding this thesis, we provide an outlook for future work that should be conducted to further progress in the research direction of socio-technical aspects of OSS projects.

8.1 Summary of the Contributions

In essence, based on the five empirical studies that we have presented in this thesis, our contribution is threefold:

1. *Insights into the Evolution of Collaboration and Communication:* In the first three studies, we investigated the evolution of collaboration and communication from three different points of view:
   a) We investigated synchronous development on different levels of coordination, that is, we analyzed whether developers collaborate and communicate temporally close-by and whether their communication is content-wise related to their joint coding activity. Therefore, we developed a new method for raising the abstraction level of communication from a message-based model to a conversation-based model. Moreover, we introduced a continuous variable *synchronicity degree* to quantify the significance of co-editing artifacts, as well as an upper bound and a lower bound for determining whether e-mail communication is related to co-editing activity. In an extensive empirical study, we observed that a more abstract, higher-level perspective

on communication captures the coordination activities of developers more accurately than a sole message-based perspective. Beside quantitative analyses, we qualitatively investigated whether e-mail communication is content-wise related to temporally close-by collaboration activities and found that only between 29% and 47% of temporally aligned collaboration and communication activities are content-wise related.

b) In our second study, we shifted the perspective on OSS projects from an activity-based view to a multi-modal network view. That is, we examined how collaboration and communication activities can be explained by each other, whether developers form stable group structures, and how these group structures evolve over time. Therefore, we devised a modular, multi-step method based on canonical tensor decomposition in combination with a state-space trend extrapolation to detect developer groups using established community-detection algorithms. In addition, we predicted the behavior of the detected developer groups using forecast algorithms. Moreover, we proposed a method to track changes and irregularities in such group structures, which could be beneficial for project stakeholders to identify potential problems in the organizational structure of the project. By means of an empirical study, we provided insights into the existence and stability of group structures in OSS projects. We were able to identify stable groups as well as distortions in the group structure. The extent to which group structures in collaboration and communication overlap is largely project specific. As our prediction of activities indicated, some group structures seem to be of changing importance, while others stayed consistently important.

c) Our third study centered around the concept of network hierarchy and a hybrid network structure that consists of a hierarchical and a non-hierarchical part. In particular, we proposed an automated method to detect hierarchical structures in a developer network and identified developers in the hierarchical as well as developers in the non-hierarchical part. Using this method, we reported insights from a longitudinal empirical study regarding the presence and evolution of hierarchical and hybrid organizational structures in OSS projects. Notably, we observed different patterns on how individual developers moved within the hierarchical structure during project evolution. Furthermore, we compared the activities and interaction partners of the developers from hierarchical and non-hierarchical parts, and we developed hypotheses regarding developers' roles within the hierarchical structure that shall guide future research on the organizational structure of OSS projects.

All three studies devised new methods and provided empirical insights that contribute toward a better understanding of how collaboration and communication activities of developers are related to each other and how developers in OSS projects organize and interact within the developer network.

2. *Means for Identifying Core Developers:* In our fourth study, we shifted our focus to different developer roles. In this regard, we devised a method to automatically identify a set of core developers based on privileged role permissions of events that are triggered by the developers in GitHub issues and pull requests. By means of a validation study, we validated our automatically derived set of privileged developers with publicly available maintainer lists, resulting in that our derived set is sound (but not necessarily complete, as not all privileged developers make use of their privileges regularly). Moreover, as

our method cannot be used to determine potential future core developers before they receive privileges, we performed an assessment of the accuracy of several state-of-the-art unsupervised developer-role classification methods to help researchers and practitioners to choose an appropriate classification method. In addition, we conducted an explorative experiment to investigate how our automatic method of core-developer identification can be used to train supervised classification methods. Our empirical study revealed that the choice of data source (collaboration on the source code vs. communication) matters more than the actual classification metric.

3. *Insights into the Relation Between Network Characteristics and Organizational Events:* In our fifth study, we investigated whether organizational events in OSS projects (e.g., company acquisitions, fork creations, or technology switches) are reflected in particular characteristics of developer networks. While this study was explorative and of a rather preliminary nature, we collected various insights into the evolution of network characteristics of developer networks and their potential relation to organizational events. Based on our observations from eight case studies, we developed hypotheses regarding which network characteristics might be promising indicators for detecting organizational events that affect the structure of developer networks. This way, we lay the foundation for future work on understanding the causes and effects of organizational events, as well as for developing measures to detect potentially critical events and for developing countermeasures to avoid potentially negative consequences for the project's sustainability.

All in all, this thesis contributes toward a better understanding of the organizational structures in OSS projects, considering the evolution of a project as well as the interactions among humans in a technical setting. Our results are of practical value, for researchers and practitioners: While researchers can build upon our studies and methods to develop mechanisms to improve coordination processes, which eventually shall lead to an improvement of software quality, practitioners (e.g., project managers) can use the insights from our studies to identify sources of problems and develop ideas on how to mitigate them.

8.2 Future Work

As already indicated in the individual chapters of this thesis, our research needs to be continued in various ways to fully harness its potential. Beside the individual suggestions on how our work should be continued, which we have discussed already in the individual chapters, we provide a selection of more general ideas on future work on the analysis of emerging organizational patterns in evolving OSS projects.

Human Aspects The creation and maintenance of software highly depends on the interactions among different human beings, as we have shown in this thesis. As already indicated by some of the organizational events that we investigated in Chapter 7, the behavior of developers and code-of-conduct violations can lead to serious problems and personal discussions in OSS projects. For example, aggressive developer behavior could lead to developers abandoning the project [JYF24] (going along with knowledge loss) or distorting the organizational structure of a project (threatening the project's success). Therefore, it is inevitable to investigate more carefully how individual developers' well-being within the community is related to project success and sustainability. However, this is challenging, as humans' perception of emotions or sentiment is individual and subjective. Studying different perceptions may help enlightening the role of social factors in software engineering.

Combining Theoretical Concepts and Practical Application A second idea we would like to direct to future work concerns the combination of theoretical concepts and practical applications. While we have devised numerous methods for the analysis of developer coordination in OSS projects, which we also applied to numerous OSS projects in our empirical studies, our work lacks a practical application that makes our insights and contributions available to actual project managers and software developers. In particular, we suggest that future work should concentrate on developing dashboards and other technical means that help project stakeholders to improve coordination and prevent activities that may harm the project based on the insights from our studies, such as the early-warning mechanisms that we have mentioned in Chapter 7.

Technological Advancement Finally, even technological progress does not stop at software development. Tools that are based on artificial intelligence might advance software development into new spheres. The methods and mechanisms that we have provided in our thesis (e.g., developer-role classification methods or methods to identify structural changes and irregularities in developer networks) could profit from automated learning techniques, applied on huge datasets of OSS ecosystems, and from instant application to guide developers in their development and coordination activities.

With the emergence of artificial-intelligence-assisted software-development practices in recent years (e.g., GitHub Copilot [NN22] or ChatGPT [NPD23]), we expect that novel collaboration and communication structures emerge in OSS projects. Thus, as developers are supported by non-human actors (e.g., bots) to develop and maintain software, it is inevitable to investigate the interactions between human and non-human actors and how they affect the organizational structure and the developer networks of OSS projects—with respect to collaboration and communication activities. Also, the roles and tasks developers take in OSS projects may be subject to change when non-human actors become more and more involved in common software-development processes. Therefore, future work needs to analyze how the behavior of developers in OSS projects changes in consequence of the involvement of non-human actors. For this purpose, future work can build upon the various methods and insights that we have presented in this thesis. In summary, there are many starting points on how to continue our research with practical value for the OSS community.

Having laid out ideas for future work, we conclude that further research on emerging organizational patterns in evolving OSS projects will be beneficial for the improvement of coordination processes in OSS projects, which eventually shall serve toward less flawed software products. This thesis lays the foundations for that by enabling and obtaining a better understanding of developer activity and coordination in OSS projects and opens the door for the development of new methods to improve the organizational structures of OSS projects.

Appendix A

In this thesis, we have presented five different empirical studies to analyze developer activity and coordination with respect to emerging organizational patterns in evolving open-source software (OSS) projects. For each of the studies, we only have presented a selection of the results, due to the high number of investigations that we have made and plots that we have produced in each of our studies. Nevertheless, we provide supplementary material for each of our studies on an individual supplementary website for each study, also including replication packages, to enable traceability and reproducibility of our results. In what follows, we provide an overview of all the supplementary material that is already referenced in the individual chapters and refer to the corresponding websites.

Supplementary Material for Chapter 3

A replication package for our study on synchronous development in OSS projects, including pseudonymized raw data, analysis scripts, and results, is available on the following supplementary website:

`https://se-sic.github.io/paper-coordination-bursts/` and
`https://zenodo.org/record/5131282`.

Note that the way in which the replication package is presented on the supplementary website slightly deviates from the content that we have presented in this thesis, since the website belongs to Bock et al. [BHJ+22], which also covers additional research questions that are out of scope of this thesis. Nevertheless, beside presenting additional data and analyses that are not present in this thesis, the results on the supplementary website and the ones that we have presented in this thesis are identical.

Supplementary Material for Chapter 4

A replication package for our study on group dynamics and group stability in OSS projects, including analysis scripts, descriptions on how to gather the corresponding raw data, and results, is available on the following supplementary website:

`https://se-sic.github.io/paper-groupdynamics-oss-tensordecomposition/` and
`https://zenodo.org/records/10556710`.

Supplementary Material for Chapter 5

A replication package for our study on hybrid and hierarchical structures in OSS projects, including analysis scripts, pseudonymized raw data, and results, is available on the following supplementary website:
`https://hierarchypaper.bitbucket.io/public/` and
`https://zenodo.org/record/7199267`.

Supplementary Material for Chapter 6

A replication package for our study on identifying core developers in OSS projects, including pseudonymized raw data, classification data, tooling, analysis scripts, and results, is available on the following supplementary website:
`https://se-sic.github.io/paper-developer-classifications/` and
`https://zenodo.org/record/7775882`.

Supplementary Material for Chapter 7

A replication package for our study on organizational events in OSS projects, including pseudonymized raw data, analysis scripts, and results, is available on the following supplementary website:
`https://se-sic.github.io/website-event-analysis/` and
`https://zenodo.org/records/10614399`.

Bibliography

[AB15] Bilal Afsar and Yuosre F. Badir. “The Impacts of Person-Organisation Fit and Perceived Organisational Support on Innovative Work Behaviour: The Mediating Effects of Knowledge Sharing Behaviour.” In: *International Journal of Information Systems and Change Management* (*IJISCM*) 7.4 (2015), pp. 263–285.

[ABF+08] Edoardo M. Airoldi, David M. Blei, Stephen E. Fienberg, and Eric P. Xing. “Mixed Membership Stochastic Blockmodels.” In: *Journal of Machine Learning Research* (*JMLR*) 9.65 (2008), pp. 1981–2014.

[ABW+13] Ban Al-Ani, Matthew J. Bietz, Yi Wang, Erik Trainer, Benjamin Koehne, Sabrina Marczak, David Redmiles, and Rafael Prikladnicki. “Globally Distributed System Developers: Their Trust Expectations and Processes.” In: *Proc. Int. Conf. Computer-Supported Cooperative Work* (*CSCW*). ACM, 2013, pp. 563–574.

[ACM+21] Zeinab Abou Khalil, Eleni Constantinou, Tom Mens, and Laurence Duchien. “On the Impact of Release Policies on Bug Handling Activity: A Case Study of Eclipse.” In: *Journal of Systems and Software* (*JSS*) 173, 110882 (2021).

[AFH+12] Anima Anandkumar, Dean P. Foster, Daniel J. Hsu, Sham M. Kakade, and Yi-Kai Liu. “A Spectral Algorithm for Latent Dirichlet Allocation.” In: *Advances in Neural Information Processing Systems* (*NIPS*). Curran Associates, Inc., 2012, pp. 926–934.

[AGA13] Nasir Ali, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. “Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links.” In: *IEEE Transactions on Software Engineering* (*TSE*) 39.5 (2013), pp. 725–741.

[AGH+14a] Animashree Anandkumar, Rong Ge, Daniel J. Hsu, and Sham M. Kakade. “A Tensor Approach to Learning Mixed Membership Community Models.” In: *Journal of Machine Learning Research* (*JMLR*) 15.1 (2014), pp. 2239–2312.

[AGH+14b] Animashree Anandkumar, Rong Ge, Daniel J. Hsu, Sham M. Kakade, and Matus Telgarsky. “Tensor Decompositions for Learning Latent Variable Models.” In: *Journal of Machine Learning Research* (*JMLR*) 15.1 (2014), pp. 2773–2832.

[AKH+16] Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, and Daniel M. Germán. “An Empirical Study of Integration Activities in Distributions of Open Source Software.” In: *Empirical Software Engineering* (*EMSE*) 21.3 (2016), pp. 960–1001.

[AKM08] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. “What’s a Typical Commit? A Characterization of Open Source Software Repositories.” In: *Proc. Int. Conf. Program Comprehension* (*ICPC*). IEEE, 2008, pp. 182–191.

[AMM+21] Usman Ashraf, Christoph Mayr-Dorn, Atif Mashkoor, Alexander Egyed, and Sebastiano Panichella. "Do Communities in Developer Interaction Networks Align with Subsystem Developer Teams? An Empirical Study of Open Source Systems." In: *Proc. Joint Int. Conf. Software and System Processes (ICSSP) and Int. Conf. Global Software Engineering (ICGSE)*. IEEE, 2021, pp. 61–71.

[APH+16] Guilherme Avelino, Leonardo Passos, André Hora, and Marco T. Valente. "A Novel Approach for Estimating Truck Factors." In: *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[ARK+18] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. "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)*. ACM, 2018, pp. 245–253.

[AS09] Brian de Alwis and Jonathan Sillito. "Why Are Software Projects Moving From Centralized to Decentralized Version Control Systems?" In: *Proc. Int. Workshop Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2009, pp. 36–39.

[AW18] Mohamed A. Aljemabi and Zhongjie Wang. "Empirical Study on the Evolution of Developer Social Networks." In: *IEEE Access* 6 (2018), pp. 51049–51060.

[Aok13] Masanao Aoki. *State Space Modeling of Time Series*. 2nd ed. Springer, 2013.

[BA99] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks." In: *Science* 286.5439 (1999), pp. 509–512.

[BAJ+23] Thomas Bock, Nils Alznauer, Mitchell Joblin, and Sven Apel. "Automatic Core-Developer Classification on GitHub: A Validation Study." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 32.6, 138 (2023).

[BB13] Alberto Bacchelli and Christian Bird. "Expectations, Outcomes, and Challenges of Modern Code Review." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[BC14] Amiangshu Bosu and Jeffrey C. Carver. "Impact of Developer Reputation on Code Review Outcomes in OSS Projects: An Empirical Investigation." In: *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2014, 10.

[BC94] Donald J. Berndt and James Clifford. "Using Dynamic Time Warping to Find Patterns in Time Series." In: *Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD)*. AAAI Press, 1994, pp. 359–370.

[BCD+12] Mario L. Bernardi, Gerardo Canfora, Giuseppe A. Di Lucca, Massimiliano Di Penta, and Damiano Distante. "Do Developers Introduce Bugs When They Do Not Communicate? The Case of Eclipse and Mozilla." In: *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 139–148.

[BDL10] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. "Are Popular Classes More Defect Prone?" In: *Proc. Int. Conf. Fundamental Approaches to Software Engineering (FASE)*. Springer, 2010, pp. 59–73.

[BE05] Ulrik Brandes and Thomas Erlebach. *Network Analysis: Methodological Foundations*. Springer, 2005.

[BEJ18] Stephen P. Borgatti, Martin G. Everett, and Jeffrey C. Johnson. *Analyzing Social Networks*. 2nd ed. Sage, 2018.

[BGD+06] Christian Bird, Alex Gourley, Premkumar T. Devanbu, Michael Gertz, and Anand Swaminathan. "Mining Email Social Networks." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. ACM, 2006, pp. 137–143.

[BGD+07] Christian Bird, Alex Gourley, Premkumar T. Devanbu, Anand Swaminathan, and Greta Hsu. "Open Borders? Immigration in Open Source Projects." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2007, 6.

[BGK+13] Sabrina E. Bailey, Sneha S. Godbole, Charles D. Knutson, and Jonathan L. Krein. "A Decade of Conway's Law: A Literature Review from 2003–2012." In: *Int. Workshop on Replication in Empirical Software Engineering Research (RESER)*. IEEE, 2013, pp. 1–14.

[BGL+18] Simon Butler, Jonas Gamalielsson, Björn Lundell, Per Jonsson, Johan Sjöberg, Anders Mattsson, Niklas Rickö, Tomas Gustavsson, Jonas Feist, Stefan Landemoo, and Erik Lönroth. "An Investigation of Work Practices Used by Companies Making Contributions to Established OSS Projects." In: *Proc. Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2018, pp. 201–210.

[BGL+21] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Johan Sjöberg, Anders Mattsson, Tomas Gustavsson, Jonas Feist, and Erik Lönroth. "On Company Contributions to Community Open Source Software Projects." In: *IEEE Transactions on Software Engineering (TSE)* 47.7 (2021), pp. 1381–1401.

[BHJ+22] Thomas Bock, Claus Hunsen, Mitchell Joblin, and Sven Apel. "Synchronous Development in Open-Source Projects: A Higher-Level Perspective." In: *Automated Software Engineering* 29.1, 3 (2022).

[BKZ10] Andrew Begel, Yit P. Khoo, and Thomas Zimmermann. "Codebook: Discovering and Exploiting Relationships in Software Repositories." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2010, pp. 125–134.

[BLJ+13] Tegawendé F. Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillere, Jacques Klein, and Yves Le Traon. "Got Issues? Who Cares About It? A Large Scale Investigation of Issue Trackers from GitHub." In: *Proc. Int. Sympos. Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 188–197.

[BMF+13] Stefanie Betz, Darja Mite, Samuel Fricker, Andrew Moss, Wasif Afzal, Mikael Svahnberg, Claes Wohlin, Jürgen Börstler, and Tony Gorschek. "An Evolutionary Perspective on Socio-Technical Congruence: The Rubber Band Effect." In: *Int. Workshop on Replication in Empirical Software Engineering Research (RESER)*. IEEE, 2013, pp. 15–24.

[BNG+09] Christian Bird, Nachiappan Nagappan, Harald C. Gall, Brendan Murphy, and Premkumar T. Devanbu. "Putting It All Together: Using Socio-Technical Networks to Predict Failures." In: *Proc. Int. Sympos. Software Reliability Engineering (ISSRE)*. IEEE, 2009, pp. 109–119.

[BNM+11] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald C. Gall, and Premkumar T. Devanbu. "Don't Touch My Code! Examining the Effects of Ownership on Software Quality." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 4–14.

[BON+22] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. "Reuse and Maintenance Practices Among Divergent Forks in Three Software Ecosystems." In: *Empirical Software Engineering (EMSE)* 27.2, 54 (2022).

[BP19] Chris Brown and Chris Parnin. "Sorry to Bother You: Designing Bots for Effective Recommendations." In: *Int. Workshop on Bots in Software Engineering (BotSE)*. IEEE, 2019, pp. 54–58.

[BPD+08] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar T. Devanbu. "Latent Social Structure in Open Source Projects." In: *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 2008, pp. 24–35.

[BR03] Andrea Bonaccorsi and Cristina Rossi. "Why Open Source Software Can Succeed." In: *Research Policy* 32.7 (2003), pp. 1243–1258.

[BSA22] Thomas Bock, Angelika Schmid, and Sven Apel. "Measuring and Modeling Group Dynamics in Open-Source Software Development: A Tensor Decomposition Approach." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2, 19 (2022).

[Bar16] Albert-László Barabási. *Network Science*. 1st ed. Cambridge University Press, 2016.

[Bir11] Christian Bird. "Sociotechnical Coordination and Collaboration in Open Source Software." In: *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 2011, pp. 568–573.

[Bon07] Phillip Bonacich. "Some Unique Properties of Eigenvector Centrality." In: *Social Networks* 29.4 (2007), pp. 555–564.

[Bon72] Phillip Bonacich. "Factoring and Weighting Approaches to Status Scores and Clique Identification." In: *Journal of Mathematical Sociology* 2.1 (1972), pp. 113–120.

[Bon87] Phillip Bonacich. "Power and Centrality: A Family of Measures." In: *American Journal of Sociology* 92.5 (1987), pp. 1170–1182.

[CB16] Lyra J. Colfer and Carliss Y. Baldwin. "The Mirroring Hypothesis: Theory, Evidence, and Exceptions." In: *Industrial and Corporate Change* 25.5 (2016), pp. 709–738.

[CBH+02] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. "SMOTE: Synthetic Minority Over-Sampling Technique." In: *Journal of Artificial Intelligence Research (JAIR)* 16 (2002), pp. 321–357.

[CBO+20] Edna D. Canedo, Rodrigo Bonifácio, Márcio V. Okimoto, Alexander Serebrenik, Gustavo Pinto, and Eduardo Monteiro. "Work Practices and Perceptions from Women Core Developers in OSS Communities." In: *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2020, pp. 1–11.

[CC70] J. Douglas Carroll and Jih-Jie Chang. "Analysis of Individual Differences in Multidimensional Scaling via an N-Way Generalization of 'Eckart-Young' Decomposition." In: *Psychometrika* 35.3 (1970), pp. 283–319.

[CC79] Thomas D. Cook and Donald T. Campbell. *Quasi-Experimentation – Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, 1979.

[CCZ+19] Zitai Chen, Chuan Chen, Zibin Zheng, and Yi Zhu. "Tensor Decomposition for Multilayer Networks Clustering." In: *Proc. AAAI Conf. Artificial Intelligence (AAAI)*. AAAI Press, 2019, pp. 3371–3378.

[CDO+12] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. "Who is Going to Mentor Newcomers in Open Source Projects?" In: *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 2012, pp. 1–11.

[CEA+01] Reuven Cohen, Keren Erez, Daniel ben Avraham, and Shlomo Havlin. "Breakdown of the Internet under Intentional Attack." In: *Physical Review Letters* 86 (16 2001), pp. 3682–3685.

[CG16] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." In: *Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 2016, pp. 785–794.

[CH05] Kevin Crowston and James Howison. "The Social Structure of Free and Open Source Software Development." In: *First Monday* 10.2 (2005).

[CH06] Kevin Crowston and James Howison. "Hierarchy and Centralization in Free and Open Source Software Team Communications." In: *Knowledge, Technology & Policy* 18.4 (2006), pp. 65–85.

[CH08] Marcelo Cataldo and James D. Herbsleb. "Communication Networks in Geographically Distributed Software Development." In: *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW)*. ACM, 2008, pp. 579–588.

[CH13] Marcelo Cataldo and James D. Herbsleb. "Coordination Breakdowns and Their Impact on Development Productivity and Software Failures." In: *IEEE Transactions on Software Engineering (TSE)* 39.3 (2013), pp. 343–360.

[CHC08] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. "Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity." In: *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2008, pp. 2–11.

[CHY+18] Bolun Chen, Yong Hua, Yan Yuan, and Ying Jin. "Link Prediction on Directed Networks Based on AUC Optimization." In: *IEEE Access* 6 (2018), pp. 28122–28136.

[CK07] Jacques J. F. Commandeur and Siem J. Koopman. *An Introduction to State Space Time Series Analysis*. Practical Econometrics Series. Oxford University Press, 2007.

[CLL+17] Can Cheng, Bing Li, Zeng-Yang Li, Yu-Qi Zhao, and Feng-Ling Liao. "Developer Role Evolution in Open Source Software Ecosystem: An Explanatory Study on GNOME." In: *Journal of Computer Science and Technology* (*JCST*) 32.2 (2017), pp. 396–414.

[CLN17] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. "A Preliminary Analysis on the Effects of Propensity to Trust in Distributed Software Development." In: *Proc. Int. Conf. Global Software Engineering* (*ICGSE*). IEEE, 2017, pp. 56–60.

[CLW+07] Kevin Crowston, Qing Li, Kangning Wei, U. Yeliz Eseryel, and James Howison. "Self-Organization of Teams for Free/Libre Open Source Software Development." In: *Information and Software Technology* (*IST*) 49.6 (2007), pp. 564–575.

[CMP07] Stefano Comino, Fabio M. Manenti, and Maria L. Parisi. "From Planning to Mature: On the Success of Open Source Projects." In: *Research Policy* 36.10 (2007), pp. 1575–1586.

[CMR+09] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. "Software Dependencies, Work Dependencies, and Their Impact on Failures." In: *IEEE Transactions on Software Engineering* (*TSE*) 35.6 (2009), pp. 864–878.

[CNM04] Aaron Clauset, Mark E. J. Newman, and Cristopher Moore. "Finding Community Structure in Very Large Networks." In: *Physical Review E* 70.6, 066111 (2004).

[CPT+19] Gemma Catolino, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Filomena Ferrucci. "Gender Diversity and Women in Software Teams: How Do They Affect Community Smells?" In: *Proc. Int. Conf. Software Engineering: Software Engineering in Society* (*ICSE-SEIS*). ACM, 2019, pp. 11–20.

[CS17] Kevin Crowston and Ivan Shamshurin. "Core-Periphery Communication and the Success of Free/Libre Open Source Software Projects." In: *Journal of Internet Services and Applications* (*JISA*) 8.1, 10 (2017).

[CSC93] Janis A. Cannon-Bowers, Eduardo Salas, and Sharolyn Converse. "Shared Mental Models in Expert Team Decision Making." In: *Individual and Group Decision Making: Current Issues*. Lawrence Erlbaum Associates, 1993. Chap. 12, pp. 221–246.

[CSN09] Aaron Clauset, Cosma R. Shalizi, and Mark E. J. Newman. "Power-Law Distributions in Empirical Data." In: *SIAM Review* (*SIREV*) 51.4 (2009), pp. 661–703.

[CTB+15] Bora Caglayan, Burak Turhan, Ayse Bener, Mayy Habayeb, Andriy Miransky, and Enzo Cialini. "Merits of Organizational Metrics in Defect Prediction: An Industrial Replication." In: *Proc. Int. Conf. Software Engineering* (*ICSE*). IEEE, 2015, pp. 89–98.

[CV17] Jailton Coelho and Marco T. Valente. "Why Modern Open Source Projects Fail." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 186–196.

[CVL+22] Shobhana Chandra, Sanjeev Verma, Weng M. Lim, Satish Kumar, and Naveen Donthu. "Personalization in Personalized Marketing: Trends and Ways Forward." In: *Psychology & Marketing* 39.8 (2022), pp. 1529–1562.

[CVS+18] Jailton Coelho, Marco T. Valente, Luciana L. Silva, and André Hora. "Why We Engage in FLOSS: Answers from Core Developers." In: *Proc. Int. Workshop Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2018, pp. 114–121.

[CWH+06] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools." In: *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW)*. ACM, 2006, pp. 353–362.

[CWL+06] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. "Core and Periphery in Free/Libre and Open Source Software Team Communications." In: *Proc. Hawaii Int. Conf. System Sciences (HICSS)*. IEEE, 2006, 118a–118a.

[CZS+20] Kattiana Constantino, Shurui Zhou, Mauricio Souza, Eduardo Figueiredo, and Christian Kästner. "Understanding Collaborative Software Development: An Interview Study." In: *Proc. Int. Conf. Global Software Engineering (ICGSE)*. ACM, 2020, pp. 55–65.

[Con68] Melvin E. Conway. "How Do Committees Invent?" In: *Datamation* 14.4 (1968), pp. 28–31.

[DB05] Trung T. Dinh-Trong and James M. Bieman. "The FreeBSD Project: A Replication Case Study of Open Source Development." In: *IEEE Transactions on Software Engineering (TSE)* 31.6 (2005), pp. 481–494.

[DC13] Anna Deluca and Álvaro Corral. "Fitting and Goodness-of-Fit Test of Non-Truncated and Truncated Power-Law Distributions." In: *Acta Geophysica* 61.6 (2013), pp. 1351–1394.

[DK12] James Durbin and Siem J. Koopman. *Time Series Analysis by State Space Methods*. Oxford Statistical Science Series. Oxford University Press, 2012.

[DKK11] Daniel M. Dunlavy, Tamara G. Kolda, and W. Philip Kegelmeyer. "Multilinear Algebra for Analyzing Data with Multiple Linkages." In: *Graph Algorithms in the Language of Linear Algebra*. Software, Environment, Tools. Society for Industrial and Applied Mathematics (SIAM), 2011, pp. 85–114.

[DKS+10] Subhajit Datta, Vikrant S. Kaulgud, Vibhu S. Sharma, and Nishant Kumar. "A Social Network Based Study of Software Team Dynamics." In: *Proc. India Software Engineering Conf. (ISEC)*. ACM, 2010, pp. 33–42.

[DLK+14] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. "The Matter of Heartbleed." In: *Proc. Conf. Internet Measurement Conference (IMC)*. ACM, 2014, pp. 475–488.

[DM03] Sergei N. Dorogovtsev and José F. F. Mendes. *Evolution of Networks: From Biological Nets to the Internet and WWW*. Oxford University Press, 2003.

[DML16] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. "What Makes a Satisficing Bug Report?" In: *Proc. Int. Conf. Software Quality, Reliability and Security (QRS)*. IEEE, 2016, pp. 164–174.

[DOB+18] Giuseppe Destefanis, Marco Ortu, David Bowes, Michele Marchesi, and Roberto Tonelli. "On Measuring Affects of GitHub Issues' Commenters." In: *Proc. Int. Workshop on Emotion Awareness in Software Engineering (SEmotion)*. ACM, 2018, pp. 14–19.

[DP03] Dirk Draheim and Lukasz Pekacki. "Process-Centric Analytical Processing of Version Control Data." In: *Proc. Int. Workshop on Principles of Software Evolution (IWPSE)*. IEEE, 2003, pp. 131–136.

[DPK+20] Natarajan Deepa, Boopathy Prabadevi, L. B. Krithika, and B. Deepa. "An Analysis on Version Control Systems." In: *Int. Conf. Emerging Trends in Information Technology and Engineering (ic-ETITE)*. IEEE, 2020, pp. 1–9.

[DR21] Tianjie Deng and William N. Robinson. "Changes in Emergent Software Development Routines: The Moderation Effects of Routine Diversity." In: *International Journal of Information Management (IJIM)* 58, 102306 (2021).

[DSP18] Luis F. Dias, Igor Steinmacher, and Gustavo Pinto. "Who Drives Company-Owned OSS Projects: Internal or External Members?" In: *Journal of the Brazilian Computer Society (JBCS)* 24.1, 16 (2018).

[DST+12] Laura Dabbish, Colleen Stuart, Jason Tsay, and James D. Herbsleb. "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository." In: *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW)*. ACM, 2012, pp. 1277–1286.

[DTW+12] Yuxiao Dong, Jie Tang, Sen Wu, Jilei Tian, Nitesh V. Chawla, Jinghai Rao, and Huanhuan Cao. "Link Prediction and Recommendation Across Heterogeneous Social Networks." In: *Proc. Int. Conf. Data Mining (ICDM)*. IEEE, 2012, pp. 181–190.

[Duc05] Nicolas Ducheneaut. "Socialization in an Open Source Software Community: A Socio-Technical Analysis." In: *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW)* 14.4 (2005), pp. 323–368.

[EGW22] Verena Ebert, Daniel Graziotin, and Stefan Wagner. "How Are Communication Channels on GitHub Presented to Their Intended Audience? – A Thematic Analysis." In: *Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2022, pp. 40–49.

[EKB+17] Ikram El Asri, Noureddine Kerzazi, Lamia Benhiba, and Mohammed Janati. "From Periphery to Core: A Temporal Analysis of GitHub Contributors' Collaboration Network." In: *Proc. Working Conf. Virtual Enterprises (PRO-VE): Collaboration in a Data-Rich World*. Springer, 2017, pp. 217–229.

[EKL+01] Alberto Espinosa, Robert Kraut, Javier Lerch, Sandra Slaughter, James D. Herbsleb, and Audris Mockus. "Shared Mental Models and Coordination in Large-Scale, Distributed Software Development." In: *Proc. Int. Conf. Information Systems (ICIS)*. Association for Information Systems, 2001, pp. 513–517.

[EP91] Kweku Ewusi-Mensah and Zbigniew H. Przasnyski. "On Information Systems Project Abandonment: An Exploratory Study of Organizational Practices." In: *MIS Quarterly* 15.1 (1991), pp. 67–86.

[ER59] Paul Erdős and Alfréd Rényi. "On Random Graphs." In: *Publicationes Mathematicae* 6 (1959), pp. 290–297.

[EZZ19] Mariam El Mezouar, Feng Zhang, and Ying Zou. "An Empirical Study on the Teams Structures in Social Coding Using GitHub Projects." In: *Empirical Software Engineering (EMSE)* 24.6 (2019), pp. 3790–3823.

[Eve13] Brian S. Everitt. *An Introduction to Latent Variable Models*. Monographs on Statistics and Applied Probability. Springer, 2013.

[FFG18] Sofia Fernandes, Hadi Fanaee-T., and João Gama. "Dynamic Graph Summarization: A Tensor Decomposition Approach." In: *Data Mining and Knowledge Discovery* 32.5 (2018), pp. 1397–1420.

[FH15] Bailey K. Fosdick and Peter D. Hoff. "Testing and Modeling Dependencies Between a Network and Nodal Attributes." In: *Journal of the American Statistical Association (JASA)* 110.511 (2015), pp. 1047–1056.

[FHV23] Hongbo Fang, James D. Herbsleb, and Bogdan Vasilescu. "Matching Skills, Past Collaboration, and Limited Competition: Modeling When Open-Source Projects Attract Contributors." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2023, pp. 42–54.

[FLL+14] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 391–400.

[FMV+19] Mivian Ferreira, Thais Mombach, Marco T. Valente, and Kecia Ferreira. "Algorithms for Estimating Truck Factors: A Comparative Study." In: *Software Quality Journal* 27.4 (2019), pp. 1583–1617.

[FPB+15] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C. Murphy, and Jean-Rémy Falleri. "Impact of Developer Turnover on Quality in Open-Source Software." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 829–841.

[FSV20] Fabio Ferreira, Luciana L. Silva, and Marco T. Valente. "Turnover in Open-Source Projects: The Case of Core Developers." In: *Proc. Brazilian Sympos. on Software Engineering (SBES)*. ACM, 2020, pp. 447–456.

[Far19] Mike Farjam. "On Whom Would I Want to Depend; Humans or Computers?" In: *Journal of Economic Psychology* 72 (2019), pp. 219–228.

[Fog05] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc., 2005.

[GBL+13] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. "Communication in Open Source Software Development Mailing Lists." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2013, pp. 277–286.

[GDL+21] Mehdi Golzadeh, Alexandre Decan, Damien Legay, and Tom Mens. "A Ground-Truth Dataset and Classification Model for Detecting Bots in GitHub Issue and PR Comments." In: *Journal of Systems and Software (JSS)* 175, 110911 (2021).

[GF17] Mohammad Gharehyazie and Vladimir Filkov. "Tracing Distributed Collaborative Development in Apache Software Foundation Projects." In: *Empirical Software Engineering (EMSE)* 22.4 (2017), pp. 1795–1830.

[GHI21] R. Stuart Geiger, Dorothy Howard, and Lilly Irani. "The Labor of Maintaining and Scaling Free and Open-Source Software Projects." In: *Proceedings of the ACM on Human-Computer Interaction (HCI)* 5.CSCW1, 175 (2021).

[GHJ98] Harald C. Gall, Karin Hajek, and Mehdi Jazayeri. "Detection of Logical Coupling based on Product Release History." In: *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 1998, pp. 190–198.

[GHP99] Rebecca E. Grinter, James D. Herbsleb, and Dewayne E. Perry. "The Geography of Coordination: Dealing with Distance in R&D Work." In: *Proc. Int. Conf. Supporting Group Work (GROUP)*. ACM, 1999, pp. 306–315.

[GKD+19] Mahen Gandhi, Amit Kumar, Yugandhar Desai, and Sonali Agarwal. "Studying Multifaceted Collaboration of OSS Developers and its Impact on Their Bug Fixing Performance." In: *Proc. Int. Workshop Quantitative Approaches to Software Quality (QuASoQ)*. CEUR Workshop Proceedings, 2019, pp. 37–44.

[GKR98] David Gibson, Jon Kleinberg, and Prabhakar Raghavan. "Inferring Web Communities from Link Topology." In: *Proc. Int. Conf. Hypertext and Hypermedia (HT)*. ACM, 1998, pp. 225–234.

[GL14] Jonas Gamalielsson and Björn Lundell. "Sustainability of Open Source Software Communities Beyond a Fork: How and Why Has the LibreOffice Project Evolved?" In: *Journal of Systems and Software (JSS)* 89 (2014), pp. 128–145.

[GLD+20] Mehdi Golzadeh, Damien Legay, Alexandre Decan, and Tom Mens. "Bot or Not? Detecting Bots in GitHub Pull Request Activity Based on Comment Similarity." In: *Proc. Int. Conf. Software Engineering Workshops (ICSEW)*. ACM, 2020, pp. 31–35.

[GLM06] Rajdeep Grewal, Gary L. Lilien, and Girish Mallapragada. "Location, Location, Location: How Network Embeddedness Affects Project Success in Open Source Systems." In: *Management Science* 52.7 (2006), pp. 1043–1056.

[GM07] Yongqin Gao and Greg Madey. "Network Analysis of the SourceForge.net Community." In: *Proc. IFIP Int. Conf. Open Source Systems: Open Source Development, Adoption and Innovation*. Springer, 2007, pp. 187–200.

[GM10] Mathieu Goeminne and Tom Mens. "A Framework for Analysing and Visualising Open Source Software Ecosystems." In: *Proc. Int. Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution (IWPSE-EVOL)*. ACM, 2010, pp. 42–47.

[GN02] Michelle Girvan and Mark E. J. Newman. "Community Structure in Social and Biological Networks." In: *Proceedings of the National Academy of Sciences (PNAS)* 99.12 (2002), pp. 7821–7826.

[GNL16] Felipe Grando, Diego Noble, and Luis C. Lamb. "An Analysis of Centrality Measures for Complex and Social Networks." In: *Global Communications Conf. (GLOBECOM)*. IEEE, 2016, pp. 1–6.

[GPC14] Laetitia Gauvin, André Panisson, and Ciro Cattuto. "Detecting the Community Structure and Activity Patterns of Temporal Networks: A Non-Negative Tensor Factorization Approach." In: *PLOS ONE* 9.1, e86028 (2014).

[GPD14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. "An Exploratory Study of the Pull-Based Software Development Model." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 345–355.

[GSS21] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. "Analysing Time-Stamped Co-Editing Networks in Software Development Teams Using git2net." In: *Empirical Software Engineering (EMSE)* 26.4, 75 (2021).

[GWR+14] Sandra González-Bailón, Ning Wang, Alejandro Rivero, Javier Borge-Holthoefer, and Yamir Moreno. "Assessing the Bias in Samples of Large Online Networks." In: *Social Networks* 38 (2014), pp. 16–27.

[Gio09] Toni Giorgino. "Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package." In: *Journal of Statistical Software* 31.7 (2009), pp. 1–24.

[HAB+18] Rob J. Hyndman, George Athanasopoulos, Christoph Bergmeir, Gabriel Caceres, Leanne Chhay, Mitchell O'Hara-Wild, Fotios Petropoulos, Slava Razbash, Earo Wang, and Farah Yasmeen. *`forecast`: Forecasting Functions for Time Series and Linear Models*. R package version 8.4. 2018.

[HAT+21] Steffen Herbold, Aynur Amirfallah, Fabian Trautsch, and Jens Grabowski. "A Systematic Mapping Study of Developer Social Network Research." In: *Journal of Systems and Software (JSS)* 171, 110802 (2021).

[HC14] James Howison and Kevin Crowston. "Collaboration Through Open Superposition: A Theory of the Open Source Way." In: *MIS Quarterly* 38.1 (2014), pp. 29–50.

[HCQ+22] Hong Huang, Jian Cao, Qing Qi, and Boxuan Zhao. "DOCEM: A Domain-Embedding-Based Open-Source Community Event Monitoring Model." In: *Computer Supported Cooperative Work and Social Computing (ChineseCSCW 2021)*. Springer, 2022, pp. 403–417.

[HDB18] Bruno C. Honorio, João P. L. De Carvalho, and Alexandro J. Baldassin. "On the Efficiency of Transactional Code Generation: A GCC Case Study." In: *Workshop of Computer Systems and High Performance (WSCAD)*. IEEE, 2018, pp. 184–190.

[HG08] Mark D. Humphries and Kevin Gurney. "Network 'Small-World-Ness': A Quantitative Method for Determining Canonical Network Equivalence." In: *PLOS ONE* 3.4, e0002051 (2008).

[HG99] James D. Herbsleb and Rebecca E. Grinter. "Architectures, Coordination, and Distance: Conway's Law and Beyond." In: *IEEE Software* 16.5 (1999), pp. 63–70.

[HGF+22] Mahsa Hadian, Soude Ghari, Marios Fokaefs, Scott Brisson, Ehsan Noei, Kelly Lyons, Bram Adams, and Shurui Zhou. "Exploring Trends and Practices of Forks in Open-Source Software Repositories." In: *Proc. Annual Int. Conf. Computer Science and Software Engineering* (*CASCON*). IBM, 2022, pp. 120–129.

[HHB+18] Marvin Hanisch, Carolin Haeussler, Stefan Berreiter, and Sven Apel. "Developers' Progression from Periphery to Core in the Linux Kernel Development Project." In: *Academy of Management Proceedings* 2018.1, 14263 (2018).

[HIC06] James Howison, Keisuke Inoue, and Kevin Crowston. "Social Dynamics of Free and Open Source Team Communications." In: *Int. Conf. Open Source Systems* (*OSS*). Springer, 2006, pp. 319–330.

[HKA+23] Shehzad Haider, Wajeeha Khalil, Ahmad S. Al-Shamayleh, Adnan Akhunzada, and Abdullah Gani. "Risk Factors and Practices for the Development of Open Source Software From Developers' Perspective." In: *IEEE Access* 11 (2023), pp. 63333–63350.

[HKC+11] Qiaona Hong, Sunghun Kim, Shing Chi Cheung, and Christian Bird. "Understanding a Developer Social Network and its Evolution." In: *Proc. Int. Conf. Software Maintenance* (*ICSM*). IEEE, 2011, pp. 323–332.

[HKS+02] Rob J. Hyndman, Anne B. Koehler, Ralph D. Snyder, and Simone Grose. "A State Space Framework for Automatic Forecasting Using Exponential Smoothing Methods." In: *International Journal of Forecasting* 18.3 (2002), pp. 439–454.

[HM03a] James D. Herbsleb and Audris Mockus. "An Empirical Study of Speed and Communication in Globally Distributed Software Development." In: *IEEE Transactions on Software Engineering* (*TSE*) 29.6 (2003), pp. 481–494.

[HM03b] James D. Herbsleb and Audris Mockus. "Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering* (*ESEC/FSE*). ACM, 2003, pp. 138–147.

[HM06] Pamela J. Hinds and Cathleen McGrath. "Structures That Work: Social Structure, Work Structure and Coordination Ease in Geographically Distributed Teams." In: *Proc. Int. Conf. Computer-Supported Cooperative Work* (*CSCW*). ACM, 2006, pp. 343–352.

[HMR06] James D. Herbsleb, Audris Mockus, and Jeffrey A. Roberts. "Collaboration in Software Engineering Projects: A Theory of Coordination." In: *Proc. Int. Conf. Information Systems* (*ICIS*). Association for Information Systems, 2006, pp. 553–568.

[HR09] Andrea Hemetsberger and Christian Reinhardt. "Collective Development in Open-Source Communities: An Activity Theoretical Perspective on Successful Online Collaboration." In: *Organization Studies* 30.9 (2009), pp. 987–1008.

[HR17] Shuk Y. Ho and Arun Rai. "Continued Voluntary Participation Intention in Firm-Participating Open Source Software Projects." In: *Information Systems Research* (*ISR*) 28.3 (2017), pp. 603–625.

[HRH02] Peter D. Hoff, Adrian E. Raftery, and Mark S. Handcock. "Latent Space Approaches to Social Network Analysis." In: *Journal of the American Statistical Association* (*JASA*) 97.460 (2002), pp. 1090–1098.

[HSA20] Claus Hunsen, Janet Siegmund, and Sven Apel. "On the Fulfillment of Coordination Requirements in Open-Source Software Projects: An Exploratory Study." In: *Empirical Software Engineering* (*EMSE*) 25.6 (2020), pp. 4379–4426.

[HZ13] Petra Heck and Andy Zaidman. "An Analysis of Requirements Evolution in Open Source Projects: Recommendations for Issue Trackers." In: *Proc. Int. Workshop on Principles of Software Evolution* (*IWPSE*). ACM, 2013, pp. 43–52.

[Har70] Richard A. Harshman. "Foundations of the PARAFAC Procedure: Models and Conditions for an 'Explanatory' Multimodal Factor Analysis." In: *UCLA Working Papers in Phonetics* 16 (1970), pp. 1–84.

[Her07] James D. Herbsleb. "Global Software Engineering: The Future of Socio-Technical Coordination." In: *Future of Software Engineering* (*FOSE*). IEEE, 2007, pp. 188–198.

[Hit27] Frank L. Hitchcock. "The Expression of a Tensor or a Polyadic as a Sum of Products." In: *Journal of Mathematics and Physics* 6.1–4 (1927), pp. 164–189.

[Hof07] Peter D. Hoff. "Modeling Homophily and Stochastic Equivalence in Symmetric Relational Data." In: *Advances in Neural Information Processing Systems* (*NIPS*). Curran Associates, Inc., 2007, pp. 657–664.

[Hof09] Peter D. Hoff. "Multiplicative Latent Factor Models for Description and Prediction of Social Networks." In: *Computational and Mathematical Organization Theory* 15.4 (2009), pp. 261–272.

[Hof11] Peter D. Hoff. "Hierarchical Multilinear Models for Multiway Data." In: *Computational Statistics & Data Analysis* 55.1 (2011), pp. 530–543.

[IB16] Jacopo Iacovacci and Ginestra Bianconi. "Extracting Information from Multiplex Networks." In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 26.6, 065306 (2016).

[IRO+09] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesús M. González-Barahona. "Using Software Archaeology to Measure Knowledge Loss in Software Projects Due to Developer Turnover." In: *Proc. Hawaii Int. Conf. System Sciences* (*HICSS*). IEEE, 2009, pp. 1–10.

[IRP+17] Martin Innes, Colin Roberts, Alun Preece, and David Rogers. "After Woolwich: Analyzing Open Source Communications to Understand the Interactive and Multi-polar Dynamics of the Arc of Conflict." In: *The British Journal of Criminology* (*BJC*) 58.2 (2017), pp. 434–454.

[JA22] Mitchell Joblin and Sven Apel. "How Do Successful and Failed Projects Differ? A Socio-Technical Analysis." In: *ACM Transactions on Software Engineering and Methodology* (*TOSEM*) 31.4, 67 (2022).

[JAH+17] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics." In: *Proc. Int. Conf. Software Engineering* (*ICSE*). IEEE, 2017, pp. 164–174.

[JAK+14] Yujuan Jiang, Bram Adams, Foutse Khomh, and Daniel M. Germán. "Tracing Back the History of Commits in Low-Tech Reviewing Environments: A Case Study of the Linux Kernel." In: *Proc. Int. Sympos. Empirical Software Engineering and Measurement* (*ESEM*). ACM, 2014, 51.

[JAM17] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary Trends of Developer Coordination: A Network Approach." In: *Empirical Software Engineering* (*EMSE*) 22.4 (2017), pp. 2050–2094.

[JEB+23] Mitchell Joblin, Barbara Eckl-Ganser, Thomas Bock, Angelika Schmid, Janet Siegmund, and Sven Apel. "Hierarchical and Hybrid Organizational Structures in Open-Source Software Projects: A Longitudinal Study." In: *ACM Transactions on Software Engineering and Methodology* (*TOSEM*) 32.4, 86 (2023).

[JLH+17] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet S. Kochhar, and Li Zhang. "Why and How Developers Fork What from Whom in GitHub." In: *Empirical Software Engineering* (*EMSE*) 22.1 (2017), pp. 547–578.

[JMA+15] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. "From Developer Networks to Verified Communities: A Fine-Grained Approach." In: *Proc. Int. Conf. Software Engineering* (*ICSE*). IEEE, 2015, pp. 563–573.

[JNB03] Hawoong Jeong, Zoltan Néda, and Albert-László Barabási. "Measuring Preferential Attachment in Evolving Networks." In: *Europhysics Letters* (*EPL*) 61.4 (2003), pp. 567–572.

[JPW17] Andrew T. Jebb, Scott Parrigon, and Sang E. Woo. "Exploratory Data Analysis as a Foundation of Inductive Research." In: *Human Resource Management Review* (*HRMR*) 27.2 (2017), pp. 265–276.

[JS05] Chris Jensen and Walt Scacchi. "Collaboration, Leadership, Control, and Conflict Negotiation and the Netbeans.org Open Source Software Development Community." In: *Proc. Hawaii Int. Conf. System Sciences* (*HICSS*). IEEE, 2005, 196b.

[JSS11] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. "Mining and Visualizing Developer Networks from Version Control Systems." In: *Proc. Int. Workshop Cooperative and Human Aspects of Software Engineering* (*CHASE*). ACM, 2011, pp. 24–31.

[JSW11] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. "The Onion Patch: Migration in Open Source Ecosystems." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering* (*ESEC/FSE*). ACM, 2011, pp. 70–80.

[JYF24] Jack Jamieson, Naomi Yamashita, and Eureka Foong. "Predicting Open Source Contributor Turnover from Value-Related Discussions: An Analysis of GitHub Issues." In: *Proc. Int. Conf. Software Engineering* (*ICSE*). ACM, 2024, pp. 667–679.

[Job17] Mitchell Joblin. "Structural and Evolutionary Analysis of Developer Networks." PhD thesis. University of Passau, 2017.

[Jol02] Ian T. Jolliffe. *Principal Component Analysis*. 2nd ed. Springer Series in Statistics. Springer, 2002.

[KAD+15] Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. "Understanding the Impact of Rapid Releases on Software Quality." In: *Empirical Software Engineering* (*EMSE*) 20.2 (2015), pp. 336–373.

[KB09] Tamara G. Kolda and Brett W. Bader. "Tensor Decompositions and Applications." In: *SIAM Review* (*SIREV*) 51.3 (2009), pp. 455–500.

[KBK05] Tamara G. Kolda, Brett W. Bader, and Joseph P. Kenny. "Higher-Order Web Link Analysis Using Multilinear Algebra." In: *Proc. Int. Conf. Data Mining* (*ICDM*). IEEE, 2005, pp. 242–249.

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. "Matrix Factorization Techniques for Recommender Systems." In: *Computer* 42.8 (2009), pp. 30–37.

[KC18] Steve W. J. Kozlowski and Georgia T. Chao. "Unpacking Team Process Dynamics and Emergent Phenomena: Challenges, Conceptual Advances, and Innovative Methods." In: *American Psychologist* 73.4 (2018), pp. 576–592.

[KCD12] Irwin Kwan, Marcelo Cataldo, and Daniela Damian. "Conway's Law Revisited: The Evidence for a Task-Based Perspective." In: *IEEE Software* 29.1 (2012), pp. 90–93.

[KFB10] Jérôme Kunegis, Damien Fay, and Christian Bauckhage. "Network Growth and the Spectral Evolution Model." In: *Proc. Int. Conf. Information and Knowledge Management* (*CIKM*). ACM, 2010, pp. 739–748.

[KG13] Amit Kumar and Avdhesh Gupta. "Evolution of Developer Social Network and Its Impact on Bug Fixing Process." In: *Proc. India Software Engineering Conf.* (*ISEC*). ACM, 2013, pp. 63–72.

[KGB+16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela Damian. "An In-Depth Study of the Promises and Perils of Mining GitHub." In: *Empirical Software Engineering* (*EMSE*) 21.5 (2016), pp. 2035–2071.

[KGB+18] Verena Käfer, Daniel Graziotin, Ivan Bogicevic, Stefan Wagner, and Jasmin Ramadani. "Communication in Open-Source Projects – End of the E-Mail Era?" In: *Proc. Int. Conf. Software Engineering: Companion Proceedings* (*ICSE Companion*). ACM, 2018, pp. 242–243.

[KJR+16] Rajiv Krishnamurthy, Varghese Jacob, Suresh Radhakrishnan, and Kutsal Dogan. “Peripheral Developer Participation in Open Source Projects: An Empirical Analysis.” In: *ACM Transactions on Management Information Systems (TMIS)* 6.4, 14 (2016).

[KP01] Eamonn J. Keogh and Michael J. Pazzani. “Derivative Dynamic Time Warping.” In: *Proc. SIAM Int. Conf. Data Mining (SDM)*. Society for Industrial and Applied Mathematics (SIAM), 2001, pp. 1–11.

[KS02] Stefan Koch and Georg Schneider. “Effort, Co-Operation and Co-Ordination in an Open Source Software Project: GNOME.” In: *Information Systems Journal (ISJ)* 12.1 (2002), pp. 27–42.

[KS95] Robert E. Kraut and Lynn A. Streeter. “Coordination in Software Development.” In: *Communications of the ACM* 38.3 (1995), pp. 69–82.

[KSD11] Irwin Kwan, Adrian Schroter, and Daniela Damian. “Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project.” In: *IEEE Transactions on Software Engineering (TSE)* 37.3 (2011), pp. 307–324.

[KT11] Ali Koc and Abdullah Uz Tansel. “A Survey of Version Control Systems.” In: *Proc. Int. Conf. Engineering and Meta-Engineering (ICEME)*. IIIS, 2011, pp. 1–6.

[KU12] Murtaza A. Khan and Faizan UrRehman. “Free and Open Source Software: Evolution, Benefits and Characteristics.” In: *Int. Journal of Emerging Trends & Technology in Computer Science (IJETTCS)* 1.3 (2012), pp. 1–7.

[Kam19] Mariusz Kamola. “How to Verify Conway’s Law for Open Source Projects.” In: *IEEE Access* 7 (2019), pp. 38469–38480.

[Kar13] Siim Karus. “Automatic Means of Identifying Evolutionary Events in Software Development.” In: *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 2013, pp. 412–415.

[Kie00] Henk A. L. Kiers. “Towards a Standardized Notation and Terminology in Multiway Analysis.” In: *Journal of Chemometrics: A Journal of the Chemometrics Society* 14.3 (2000), pp. 105–122.

[LBG+16] Aron Lindberg, Nicholas Berente, James Gaskin, and Kalle Lyytinen. “Coordinating Interdependencies in Online Communities: A Study of an Open Source Software Project.” In: *Information Systems Research (ISR)* 27.4 (2016), pp. 751–772.

[LBK+08] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. “Microscopic Evolution of Social Networks.” In: *Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 2008, pp. 462–470.

[LBW15] James Li, Jacob Bien, and Martin Wells. *`rTensor`: Tools for Tensor Analysis and Decomposition*. `R` package version 1.3. 2015.

[LC03] Gwendolyn K. Lee and Robert E. Cole. “From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development.” In: *Organization Science* 14.6 (2003), pp. 633–649.

[LC17] Amanda Lee and Jeffrey C. Carver. "Are One-Time Contributors Different? A Comparison to Core and Periphery Developers in FLOSS Repositories." In: *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 1–10.

[LCB17] Amanda Lee, Jeffrey C. Carver, and Amiangshu Bosu. "Understanding the Impressions, Motivations, and Barriers of One Time Code Contributors to FLOSS Projects: A Survey." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2017, pp. 187–197.

[LCQ21] Lingjia Li, Jian Cao, and Qin Qi. "Monitoring Negative Sentiment-Related Events in Open Source Software Projects." In: *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*. IEEE, 2021, pp. 92–100.

[LHH17] Manlu Liu, Clyde E. Hull, and Yu-Ting C. Hung. "Starting Open-Source Collaborative Innovation: The Antecedents of Network Formation in Community Source." In: *Information Systems Journal (ISJ)* 27.5 (2017), pp. 643–670.

[LJR08] Jorge M. Lobo, Alberto Jiménez-Valverde, and Raimundo Real. "AUC: A Misleading Measure of the Performance of Predictive Distribution Models." In: *Global Ecology and Biogeography* 17.2 (2008), pp. 145–151.

[LK06] Dorothy E. Leidner and Timothy Kayworth. "A Review of Culture in Information Systems Research: Toward a Theory of Information Technology Culture Conflict." In: *MIS Quaterly* 30.2 (2006), pp. 357–399.

[LKR12] Dean Lusher, Johan Koskinen, and Garry Robins. *Exponential Random Graph Models for Social Networks: Theory, Methods, and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, 2012.

[LM06] Amy N. Langville and Carl D. Meyer. *Google's PageRank and Beyond*. Princeton University Press, 2006.

[LM13] Sherlock A. Licorish and Stephen G. MacDonell. "The True Role of Active Communicators: An Empirical Study of Jazz Core Developers." In: *Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2013, pp. 228–239.

[LM14] Sherlock A. Licorish and Stephen G. MacDonell. "Understanding the Attitudes, Knowledge Sharing Behaviors and Task Performance of Core Developers: A Longitudinal Study." In: *Information and Software Technology (IST)* 56.12 (2014), pp. 1578–1596.

[LMV+14] Timo O. A. Lehtinen, Mika V. Mäntylä, Jari Vanhanen, Juha Itkonen, and Casper Lassenius. "Perceived Causes of Software Project Failures—An Analysis of Their Relationships." In: *Information and Software Technology (IST)* 56.6 (2014), pp. 623–643.

[LRG04] Luis López-Fernández, Gregorio Robles, and Jesús M. González-Barahona. "Applying Social Network Analysis to the Information in CVS Repositories." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE. 2004, pp. 101–105.

[LRG+06] Luis López-Fernández, Gregorio Robles, Jesús M. González-Barahona, and Israel Herraiz. "Applying Social Network Analysis Techniques to Community-Driven Libre Software Projects." In: *International Journal of Information Technology and Web Engineering (IJITWE)* 1 (3 2006), pp. 27–48.

[LRS17] Bin Lin, Gregorio Robles, and Alexander Serebrenik. "Developer Turnover in Global, Industrial Open Source Projects: Insights from Applying Survival Analysis." In: *Proc. Int. Conf. Global Software Engineering (ICGSE)*. IEEE, 2017, pp. 66–75.

[LS07] Yuan Long and Keng Siau. "Social Network Structures in Open Source Software Development Teams." In: *Journal of Database Management (JDM)* 18.2 (2007), pp. 25–40.

[LX23] Ruonan Li and Luo Xiao. "Latent Factor Model for Multivariate Functional Data." In: *Biometrics* 79.4 (2023), pp. 3307–3318.

[Lee12] Viktor Lee. *How Firms Can Strategically Influence Open Source Communities: The Employment of 'Men on the Inside'*. Gabler, 2012.

[Lon06] Ju Long. "Understanding the Role of Core Developers in Open Source Software Development." In: *Journal of Information, Information Technology, and Organizations (JIITO)* 1.1 (2006), pp. 75–85.

[MAJ+20] Umme A. Mannan, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. "On the Relationship Between Design Discussions and Design Quality: A Case Study of Apache Projects." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 543–555.

[MBN+14] Kıvanç Muşlu, Christian Bird, Nachiappan Nagappan, and Jacek Czerwonka. "Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 334–344.

[MČ15] Domagoj Margan and Sanja Čandrlić. "The Success of Open Source Software: A Review." In: *Int. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2015, pp. 1463–1468.

[MC90] Thomas W. Malone and Kevin Crowston. "What is Coordination Theory and How Can It Help Design Cooperative Work Systems?" In: *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW)*. ACM, 1990, pp. 357–370.

[MC94] Thomas W. Malone and Kevin Crowston. "The Interdisciplinary Study of Coordination." In: *ACM Computing Surveys* 26.1 (1994), pp. 87–119.

[MCD19] Tom Mens, Marcelo Cataldo, and Daniela Damian. "The Social Developer: The Future of Software Development." In: *IEEE Software* 36.1 (2019), pp. 11–14.

[MD14] M. Rocío Martínez-Torres and M. Carmen Díaz-Fernández. "Current Issues and Research Trends on Open-Source Software Communities." In: *Technology Analysis & Strategic Management* 26.1 (2014), pp. 55–68.

[ME11] Aditya K. Menon and Charles Elkan. "Link Prediction via Matrix Factorization." In: *Joint Europ. Conf. Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*. Springer, 2011, pp. 437–452.

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. "Two Case Studies of Open Source Software Development: Apache and Mozilla." In: *ACM Transactions on Software Engineering and Methodology* (*TOSEM*) 11.3 (2002), pp. 309–346.

[MFT02] Gregory Madey, Vincent Freeh, and Renee Tynan. "The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory." In: *Proc. Am. Conf. Information Systems* (*AMCIS*). Association for Information Systems, 2002, pp. 1806–1813.

[MGJL21] Kevin McClean, Des Greer, and Anna Jurek-Loughrey. "Social Network Analysis of Open Source Software: A Review and Categorisation." In: *Information and Software Technology* (*IST*) 130, 106442 (2021).

[MHP05] Martin Michlmayr, Francis Hunt, and David Probert. "Quality Practices and Problems in Free Software Projects." In: *Int. Conf. Open Source Systems* (*OSS*). IEEE, 2005, pp. 24–28.

[MJT+22] Wolfgang Mauerer, Mitchell Joblin, Damian A. Tamburri, Carlos Paradis, Rick Kazman, and Sven Apel. "In Search of Socio-Technical Congruence: A Large-Scale Longitudinal Study." In: *IEEE Transactions on Software Engineering* (*TSE*) 48.8 (2022), pp. 3159–3184.

[MKA+16] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. "An Empirical Study of the Impact of Modern Code Review Practices on Software Quality." In: *Empirical Software Engineering* (*EMSE*) 21.5 (2016), pp. 2146–2189.

[MLB18] Marcelo Marinho, Alexandre Luna, and Sarah Beecham. "Global Software Development: Practices for Cultural Differences." In: *Proc. Int. Conf. Product-Focused Software Process Improvement* (*PROFES*). Springer, 2018, pp. 299–317.

[MLM22] Lloyd Montgomery, Clara Lüders, and Walid Maalej. "An Alternative Issue Tracking Dataset of Public Jira Repositories." In: *Proc. Int. Working Conf. Mining Software Repositories* (*MSR*). ACM, 2022, pp. 73–77.

[MS20] Poonacha K. Medappa and Shirish C. Srivastava. "Ideological Shifts in Open Source Orchestration: Examining the Influence of Licence Choice and Organisational Participation on Open Source Project Outcomes." In: *European Journal of Information Systems* (*EJIS*) 29.5 (2020), pp. 500–520.

[MTM+19] Josianne Marsan, Mathieu Templier, Patrick Marois, Bram Adams, Kevin Carillo, and Georgia L. Mopenza. "Toward Solving Social and Technical Problems in Open Source Software Ecosystems: Using Cause-and-Effect Analysis to Disentangle the Causes of Complex Problems." In: *IEEE Software* 36.1 (2019), pp. 34–41.

[MUD+19] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. "Repairnator Patches Programs Automatically." In: *Ubiquity* 2019.July, 2 (2019), 2:1–2:12.

[MVS21] João E. Montandon, Marco T. Valente, and Luciana L. Silva. "Mining the Technical Roles of GitHub Users." In: *Information and Software Technology* (*IST*) 131, 106485 (2021).

[MW11] Andrew Meneely and Laurie Williams. “Socio-Technical Developer Networks: Should We Trust Our Measurements?” In: *Proc. Int. Conf. Software Engineering* (*ICSE*). ACM, 2011, pp. 281–290.

[MWK+19] Courtney Miller, David G. Widder, Christian Kästner, and Bogdan Vasilescu. “Why Do People Give Up FLOSSing? A Study of Contributor Disengagement in Open Source.” In: *Proc. IFIP Int. Conf. Open Source Systems*. Springer, 2019, pp. 116–129.

[MWS+08] Andrew Meneely, Laurie Williams, Will Snipes, and Jason A. Osborne. “Predicting Failures with Developer Networks and Social Network Analysis.” In: *Proc. Int. Sympos. on Foundations of Software Engineering* (*FSE*). ACM, 2008, pp. 13–23.

[Moc10] Audris Mockus. “Organizational Volatility and its Effects on Software Defects.” In: *Proc. Int. Sympos. on Foundations of Software Engineering* (*FSE*). ACM, 2010, pp. 117–126.

[NG04] Mark E. J. Newman and Michelle Girvan. “Finding and Evaluating Community Structure in Networks.” In: *Physical Review E* 69.2, 026113 (2004).

[NKU17] Sladjana Nørskov, Peter Kesting, and John P. Ulhøi. “Deliberate Change Without Hierarchical Influence? The Case of Collaborative OSS Communities.” In: *International Journal of Organizational Analysis* (*IJOA*) 25.2 (2017), pp. 346–374.

[NM11] Linus Nyman and Tommi Mikkonen. “To Fork or Not to Fork: Fork Motivations in SourceForge Projects.” In: *International Journal of Open Source Software and Processes* (*IJOSSP*) 3.3 (2011), pp. 1–9.

[NMB08] Nachiappan Nagappan, Brendan Murphy, and Victor R. Basili. “The Influence of Organizational Structure on Software Quality.” In: *Proc. Int. Conf. Software Engineering* (*ICSE*). ACM, 2008, pp. 521–530.

[NML+12] Linus Nyman, Tommi Mikkonen, Juho Lindman, and Martin Fougère. “Perspectives on Code Forking and Sustainability in Open Source Software.” In: *Proc. IFIP Int. Conf. Open Source Systems: Long-Term Sustainability*. Springer, 2012, pp. 274–279.

[NN22] Nhan Nguyen and Sarah Nadi. “An Empirical Evaluation of GitHub Copilot’s Code Suggestions.” In: *Proc. Int. Working Conf. Mining Software Repositories* (*MSR*). ACM, 2022, pp. 1–5.

[NPD23] Nascimento Nathalia, Alencar Paulo, and Cowan Donald. “Artificial Intelligence vs. Software Engineers: An Empirical Study on Performance and Efficiency Using ChatGPT.” In: *Proc. Annual Int. Conf. Computer Science and Software Engineering* (*CASCON*). IBM, 2023, pp. 24–33.

[NR17] Mathieu Nassif and Martin P. Robillard. “Revisiting Turnover-Induced Knowledge Loss in Software Projects.” In: *Proc. Int. Conf. Software Maintenance and Evolution* (*ICSME*). IEEE, 2017, pp. 261–272.

[NYN+02] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. "Evolution Patterns of Open-Source Software Systems and Communities." In: *Proc. Int. Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2002, pp. 76–85.

[New05] Mark E. J. Newman. "Power Laws, Pareto Distributions and Zipf's Law." In: *Contemporary Physics* 46.5 (2005), pp. 323–351.

[OF07] Siobhán O'Mahony and Fabrizio Ferraro. "The Emergence of Governance in an Open Source Community." In: *Academy of Management Journal (AMJ)* 50.5 (2007), pp. 1079–1106.

[OHM+18] Marco Ortu, Tracy Hall, Michele Marchesi, Roberto Tonelli, David Bowes, and Giuseppe Destefanis. "Mining Communication Patterns in Software Development: A GitHub Analysis." In: *Proc. Int. Conf. Predicitive Models in Software Engineering (PROMISE)*. ACM, 2018, pp. 70–79.

[OJ07] Wonseok Oh and Sangyong Jeon. "Membership Herding and Network Stability in the Open Source Community: The Ising Perspective." In: *Management Science* 53.7 (2007), pp. 1086–1101.

[OSG+15] Gustavo A. Oliva, José T. da Silva, Marco A. Gerosa, Francisco W. S. Santana, Cláudia M. L. Werner, Cleidson R. B. de Souza, and Kleverton C. M. de Oliveira. "Evolving the System's Core: A Case Study on the Identification and Characterization of Key Developers in Apache Ant." In: *Computing and Informatics* 34.3 (2015), pp. 678–724.

[OSO+12] Gustavo A. Oliva, Francisco W. Santana, Kleverton C. M. de Oliveira, Cleidson R. B. de Souza, and Marco A. Gerosa. "Characterizing Key Developers: A Case Study With Apache Ant." In: *Proc. Int. Conf. Collaboration and Technology (CRIWG)*. Springer, 2012, pp. 97–112.

[PAR14] Phanish Puranam, Oliver Alexy, and Markus Reitzig. "What's "New" About New Forms of Organizing?" In: *Academy of Management Review (AMR)* 39.2 (2014), pp. 162–180.

[PBD+14] Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. "How Developers' Collaborations Identified from Different Sources Tell us About Code Changes." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 251–260.

[PD08] Mathias Pohl and Stephan Diehl. "What Dynamic Network Metrics Can Tell Us About Developer Roles." In: *Proc. Int. Workshop Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2008, pp. 81–84.

[PL05] Pascal Pons and Matthieu Latapy. "Computing Communities in Large Networks Using Random Walks." In: *Proc. Int. Sympos. on Computer and Information Sciences (ISCIS)*. Springer, 2005, pp. 284–293.

[PMS14] Rohan Padhye, Senthil Mani, and Vibha S. Sinha. "A Study of External Community Contribution to Open-Source Projects on GitHub." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. ACM, 2014, pp. 332–335.

[PSG16] Gustavo Pinto, Igor Steinmacher, and Marco A. Gerosa. "More Common Than You Think: An In-Depth Study of Casual Contributors." In: *Int. Conf. Software Analysis, Evolution, and Reengineering* (*SANER*). IEEE, 2016, pp. 112–123.

[PT21] Fabio Palomba and Damian A. Tamburri. "Predicting the Emergence of Community Smells Using Socio-Technical Metrics: A Machine-Learning Approach." In: *Journal of Systems and Software* (*JSS*) 171, 110847 (2021).

[PT22] Prahar Pandya and Saurabh Tiwari. "CORMS: A GitHub and Gerrit Based Hybrid Code Reviewer Recommendation Approach for Modern Code Review." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering* (*ESEC/FSE*). ACM, 2022, pp. 546–557.

[Par72] David L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules." In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058.

[QCT+14] Eric Quintane, Guido Conaldi, Marco Tonellato, and Alessandro Lomi. "Modeling Relational Events: A Case Study on an Open Source Software Project." In: *Organizational Research Methods* (*ORM*) 17.1 (2014), pp. 23–50.

[QST22] Khushbakht A. Qamar, Emre Sülün, and Eray Tüzün. "Taxonomy of Bug Tracking Process Smells: Perceptions of Practitioners and an Empirical Analysis." In: *Information and Software Technology* (*IST*) 150, 106972 (2022).

[RA12] Christian Rodríguez-Bustos and Jairo Aponte. "How Distributed Version Control Systems Impact Open Source Software Projects." In: *Proc. Int. Working Conf. Mining Software Repositories* (*MSR*). IEEE, 2012, pp. 36–39.

[RAN+21] Binish Raza, Rodina Ahmad, Md. Hairul N. B. M. Nasir, and Shukor S. M. Fauzi. "Socio-Technical Congruence as an Emerging Concept in Software Development: A Scientometric Analysis and Critical Literature Review." In: *IEEE Access* 9 (2021), pp. 129051–129077.

[RB03] Erzsébet Ravasz and Albert-László Barabási. "Hierarchical Organization in Complex Networks." In: *Physical Review E* 67.2, 026112 (2003).

[RBB+22] Poedjadevie K. Ramkisoen, John Businge, Brent van Bladel, Alexandre Decan, Serge Demeyer, Coen De Roover, and Foutse Khomh. "PaReco: Patched Clones and Missed Patches Among the Divergent Variants of a Software Family." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering* (*ESEC/FSE*). ACM, 2022, pp. 646–658.

[RCO19] Mehvish Rashid, Paul M. Clarke, and Rory V. O'Connor. "A Systematic Examination of Knowledge Loss in Open Source Software Projects." In: *International Journal of Information Management* (*IJIM*) 46 (2019), pp. 104–123.

[RD15] William N. Robinson and Tianjie Deng. "Data Mining Behavioral Transitions in Open Source Repositories." In: *Proc. Hawaii Int. Conf. System Sciences* (*HICSS*). IEEE, 2015, pp. 5280–5289.

[RG06] Gregorio Robles and Jesús M. González-Barahona. "Contributor Turnover in Libre Software Projects." In: *Int. Conf. Open Source Systems* (*OSS*). Springer, 2006, pp. 273–286.

[RG12] Gregorio Robles and Jesús M. González-Barahona. "A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes." In: *Proc. IFIP Int. Conf. Open Source Systems: Long-Term Sustainability*. Springer, 2012, pp. 1–14.

[RGH09] Gregorio Robles, Jesús M. González-Barahona, and Israel Herraiz. "Evolution of the Core Team of Developers in Libre Software Projects." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2009, pp. 167–170.

[RGM05] Gregorio Robles, Jesús M. González-Barahona, and Martin Michlmayr. "Evolution of Volunteer Participation in Libre Software Projects: Evidence from Debian." In: *Int. Conf. Open Source Systems (OSS)*. IEEE, 2005, pp. 100–107.

[RGS08] Peter C. Rigby, Daniel M. Germán, and Margaret-Anne Storey. "Open Source Software Peer Review Practices: A Case Study of the Apache Server." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2008, pp. 541–550.

[RH07] Peter C. Rigby and Ahmed E. Hassan. "What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2007, pp. 23–23.

[RHR+12] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.

[RJ93] Lawrence R. Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition*. Prentice-Hall, 1993.

[RLM19] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. "The List is the Process: Reliable Pre-Integration Tracking of Commits on Mailing Lists." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2019, pp. 807–818.

[RM10] Filippo Ricca and Alessandro Marchetto. "Are Heroes Common in FLOSS Projects?" In: *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2010, pp. 1–4.

[RN88] Joseph L. Rodgers and W. Alan Nicewander. "Thirteen Ways to Look at the Correlation Coefficient." In: *The American Statistician* 42.1 (1988), pp. 59–66.

[RR14] Md. Masudur Rahman and Chanchal K. Roy. "An Insight into the Pull Requests of GitHub." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. ACM, 2014, pp. 364–367.

[RSA+19] Gregorio Robles, Igor Steinmacher, Paul Adams, and Christoph Treude. "Twenty Years of Open Source Software: From Skepticism to Mainstream." In: *IEEE Software* 36.6 (2019), pp. 12–15.

[RT12] Uzma Raja and Marietta J. Tretter. "Defining and Evaluating a Measure of Open Source Project Survivability." In: *IEEE Transactions on Software Engineering (TSE)* 38.1 (2012), pp. 163–174.

[RTH+11] Xavier Robin, Natacha Turck, Alexandre Hainard, Natalia Tiberti, Frédérique Lisacek, Jean-Charles Sanchez, and Markus Müller. "`pROC`: An Open-Source Package for `R` and `S+` to Analyze and Compare ROC Curves." In: *BMC Bioinformatics* 12.1, 77 (2011).

[RU20] Mehta Reena and Kulkarni Udita. "Impact of Personalized Social Media Advertisements on Consumer Purchase Intention." In: *Annals of the University Dunarea de Jos of Galati: Fascicle: I, Economics & Applied Informatics* 26.2 (2020), pp. 15–24.

[RZD+16] Peter C. Rigby, Yue C. Zhu, Samuel M. Donadelli, and Audris Mockus. "Quantifying and Mitigating Turnover-Induced Knowledge Loss: Case Studies of Chrome and a Project at Avaya." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2016, pp. 1006–1016.

[Rie15] Dirk Riehle. "The Five Stages of Open Source Volunteering." In: *Crowdsourcing: Cloud-Based Software Development*. Ed. by Wei Li, Michael N. Huhns, Wei-Tek Tsai, and Wenjun Wu. Springer, 2015, pp. 25–38.

[Rie19] Dirk Riehle. "The Innovations of Open Source." In: *Computer* 52.4 (2019), pp. 59–63.

[SAB18] Davide Spadini, Maurício F. Aniche, and Alberto Bacchelli. "PyDriller: Python Framework for Mining Software Repositories." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 908–911.

[SAM06] Katherine J. Stewart, Anthony P. Ammeter, and Likoebe M. Maruping. "Impacts of License Choice and Organizational Sponsorship on User Interest and Development Activity in Open Source Software Projects." In: *Information Systems Research (ISR)* 17.2 (2006), pp. 126–144.

[SBA+10] Emad Shihab, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. "On the Central Role of Mailing Lists in Open Source Projects: An Exploratory Study." In: *New Frontiers in Artificial Intelligence: JSAI-isAI 2009 Workshops*. Springer, 2010, pp. 91–103.

[SC78] Hiroaki Sakoe and Seibi Chiba. "Dynamic Programming Algorithm Optimization for Spoken Word Recognition." In: *IEEE Transactions on Acoustics, Speech, and Signal Processing (TASS)* 26.1 (1978), pp. 43–49.

[SCA+12] Stephan Spiegel, Jan Clausen, Sahin Albayrak, and Jérôme Kunegis. "Link Prediction on Evolving Data Using Tensor Factorization." In: *New Frontiers in Applied Data Mining: PAKDD 2011 International Workshops*. Springer, 2012, pp. 100–110.

[SCE+08] Barbara Scozzi, Kevin Crowston, U. Yeliz Eseryel, and Qing Li. "Shared Mental Models Among Open Source Software Developers." In: *Proc. Hawaii Int. Conf. System Sciences (HICSS)*. IEEE, 2008, pp. 306–306.

[SFD05] Cleidson R. B. de Souza, Jon Froehlich, and Paul Dourish. "Seeking the Source: Software Source Code as a Social and Technical Artifact." In: *Proc. Int. Conf. Supporting Group Work (GROUP)*. ACM, 2005, pp. 197–206.

[SFS+17] A. J. Suali, Shukor S. Md. Fauzi, W. A. W. M. Sobri, and Md. Hairul N. M. Nasir. "Developers' Coordination Issues and its Impact on Software Quality: A Systematic Review." In: *Int. Conf. Science in Information Technology (ICSITech)*. IEEE, 2017, pp. 659–663.

[SGR11] Ashish Sureka, Atul Goyal, and Ayushi Rastogi. "Using Social Network Analysis for Mining Collaboration Data in a Defect Tracking System for Risk and Vulnerability Analysis." In: *Proc. India Software Engineering Conf. (ISEC)*. ACM, 2011, pp. 195–204.

[SGR14] Igor Steinmacher, Marco A. Gerosa, and David Redmiles. "Attracting, Onboarding, and Retaining Newcomer Developers in Open-Source Software Projects." In: *Workshop on Global Software Development in a CSCW Perspective*. ACM, 2014.

[SH05] Amnon Shashua and Tamir Hazan. "Non-Negative Tensor Factorization with Applications to Statistics and Computer Vision." In: *Proc. Int. Conf. Machine Learning (ICML)*. ACM, 2005, pp. 792–799.

[SH13] M. M. Mahbubul Syeed and Imed Hammouda. "Socio-Technical Congruence in OSS Projects: Exploring Conway's Law in FreeBSD." In: *Proc. IFIP Int. Conf. Open Source Systems: Quality Verification*. Springer, 2013, pp. 109–126.

[SH17] Maha Shaikh and Ola Henfridsson. "Governing Open Source Software Through Coordination Processes." In: *Information and Organization* 27.2 (2017), pp. 116–135.

[SHD12] Pratyush N. Sharma, John Hulland, and Sherae L. Daniel. "Examining Turnover in Open Source Software Projects Using Logistic Hierarchical Linear Modeling Approach." In: *Proc. IFIP Int. Conf. Open Source Systems: Long-Term Sustainability*. Springer, 2012, pp. 331–337.

[SMS11] Vibha S. Sinha, Senthil Mani, and Saurabh Sinha. "Entering the Circle of Trust: Developer Initiation as Committers in Open-Source Projects." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. ACM, 2011, pp. 133–142.

[SMS16] Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. "From Aristotle to Ringelmann: A Large-Scale Analysis of Team Productivity and Coordination in Open Source Software Projects." In: *Empirical Software Engineering (EMSE)* 21.2 (2016), pp. 642–683.

[SMW+11] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities." In: *IEEE Transactions on Software Engineering (TSE)* 37.6 (2011), pp. 772–787.

[SR03] Andrew J. Seary and William D. Richards. "Spectral Methods for Analyzing and Visualizing Networks: An Introduction." In: *Dynamic Social Network Modeling and Analysis: Workshop Summary and Papers*. National Academies Press, 2003, pp. 209–228.

[SRS+12] Pankaj Setia, Balaji Rajagopalan, Vallabh Sambamurthy, and Roger Calantone. "How Peripheral Developers Contribute to Open-Source Software Development." In: *Information Systems Research (ISR)* 23.1 (2012), pp. 144–163.

[SSF+17] Margaret-Anne Storey, Leif Singer, Fernando F. Filho, Alexey Zagalsky, and Daniel M. Germán. “How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development.” In: *IEEE Transactions on Software Engineering (TSE)* 43.2 (2017), pp. 185–204.

[SSG14] Igor Steinmacher, Marco A. G. Silva, and Marco A. Gerosa. “Barriers Faced by Newcomers to Open Source Projects: A Systematic Review.” In: *Open Source Software: Mobile Open Source Technologies*. Springer, 2014, pp. 153–163.

[SSN09] Chandrasekar Subramaniam, Ravi Sen, and Matthew L. Nelson. “Determinants of Open Source Software Project Success: A Longitudinal Study.” In: *Decision Support Systems* 46.2 (2009), pp. 576–585.

[SSR02] Srinarayan Sharma, Vijayan Sugumaran, and Balaji Rajagopalan. “A Framework for Creating Hybrid-Open Source Software Communities.” In: *Information Systems Journal (ISJ)* 12.1 (2002), pp. 7–25.

[SSS17] Pankajeshwara N. Sharma, Bastin T. R. Savarimuthu, and Nigel Stanger. “Boundary Spanners in Open Source Software Development: A Study of Python Email Archives.” In: *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*. IEEE, 2017, pp. 308–317.

[STG19] Igor Steinmacher, Christoph Treude, and Marco A. Gerosa. “Let Me In: Guidelines for the Successful Onboarding of Newcomers to Open Source Projects.” In: *IEEE Software* 36.4 (2019), pp. 41–49.

[SWF+13] Seiji Sato, Hironori Washizaki, Yoshiaki Fukazawa, Sakae Inoue, Hiroyuki Ono, Yoshiiku Hanai, and Mikihiko Yamamoto. “Effects of Organizational Changes on Product Metrics and Defects.” In: *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*. IEEE, 2013, pp. 132–139.

[SZF+20] Natalie R. Smith, Paul N. Zivich, Leah M. Frerichs, James Moody, and Allison E. Aiello. “A Guide for Choosing Community Detection Algorithms in Social Network Studies: The Question Alignment Approach.” In: *American Journal of Preventive Medicine (AJPM)* 59.4 (2020), pp. 597–605.

[Sch22] Josef Schosser. “Tensor Extrapolation: Forecasting Large-Scale Relational Data.” In: *Journal of the Operational Research Society (JORS)* 73.5 (2022), pp. 969–978.

[Sin10] Param V. Singh. “The Small-World Effect: The Influence of Macro-Level Properties of Developer Collaboration Networks on Open-Source Project Success.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.2, 6 (2010).

[Som10] Ian Sommerville. *Software Engineering*. 9th ed. Addison-Wesley, 2010.

[Spi12] Diomidis Spinellis. “Git.” In: *IEEE Software* 29.3 (2012), pp. 100–101.

[Ste10] Maarten van Steen. *Graph Theory and Complex Networks: An Introduction*. Maarten van Steen, 2010.

[TDH14] Jason Tsay, Laura Dabbish, and James D. Herbsleb. “Let’s Talk About It: Evaluating Contributions Through Discussion in GitHub.” In: *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 144–154.

[TGW+20] Bianca Trinkenreich, Mariam Guizani, Igor S. Wiese, Anita Sarma, and Igor Steinmacher. "Hidden Figures: Roles and Pathways of Successful OSS Contributors." In: *Proceedings of the ACM on Human-Computer Interaction (HCI)* 4.CSCW2, 180 (2020).

[TGW+22] Bianca Trinkenreich, Mariam Guizani, Igor S. Wiese, Tayana Conte, Marco A. Gerosa, Anita Sarma, and Igor Steinmacher. "Pots of Gold at the End of the Rainbow: What is Success for Open Source Contributors?" In: *IEEE Transactions on Software Engineering (TSE)* 48.10 (2022), pp. 3940–3953.

[TKF23] Damian A. Tamburri, Rick Kazman, and Hamed Fahimi. "On the Relationship Between Organisational Structure Patterns and Architecture in Agile Teams." In: *IEEE Transactions on Software Engineering (TSE)* 49.1 (2023), pp. 325–347.

[TKL+15] Damian A. Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. "Social Debt in Software Engineering: Insights from Industry." In: *Journal of Internet Services and Applications (JISA)* 6, 10 (2015).

[TLV13] Damian A. Tamburri, Patricia Lago, and Hans van Vliet. "Organizational Social Structures for Software Engineering." In: *ACM Computing Surveys* 46.1, 3 (2013).

[TMB10] Sergio L. Toral, M. Rocío Martínez-Torres, and Federico Barrero. "Analysis of Virtual Communities Supporting OSS Projects Using Social Network Analysis." In: *Information and Software Technology (IST)* 52.3 (2010), pp. 296–303.

[TML14] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. "Collaboration in Open-Source Projects: Myth or Reality?" In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. ACM, 2014, pp. 304–307.

[TNK+19] Jirateep Tantisuwankul, Yusuf S. Nugroho, Raula G. Kula, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. "A Topological Analysis of Communication Channels for Knowledge Sharing in Contemporary GitHub Projects." In: *Journal of Systems and Software (JSS)* 158, 110416 (2019).

[TPK19] Damian A. Tamburri, Fabio Palomba, and Rick Kazman. "Exploring Community Smells in Open-Source: An Automated Approach." In: *IEEE Transactions on Software Engineering (TSE)* 47.3 (2019), pp. 630–652.

[TPS+19] Damian A. Tamburri, Fabio Palomba, Alexander Serebrenik, and Andy Zaidman. "Discovering Community Patterns in Open-Source: A Systematic Approach and its Evaluation." In: *Empirical Software Engineering (EMSE)* 24.3 (2019), pp. 1369–1417.

[TRC10] Antonio Terceiro, Luiz R. Rios, and Christina Chavez. "An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects." In: *Proc. Brazilian Sympos. on Software Engineering (SBES)*. IEEE, 2010, pp. 21–29.

[TZF20] Xin Tan, Minghui Zhou, and Brian Fitzgerald. "Scaling Open Source Communities: An Empirical Study of the Linux Kernel." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2020, pp. 1222–1234.

[Thu31] Louis L. Thurstone. "Multiple Factor Analysis." In: *Psychological Review* 38.5 (1931), pp. 406–427.

[VBJ11] Eric Ververs, Rick van Bommel, and Slinger Jansen. "Influences on Developer Participation in the Debian Software Ecosystem." In: *Proc. Int. Conf. Management of Emergent Digital EcoSystems (MEDES)*. ACM, 2011, pp. 89–93.

[VDK76] Andrew H. Van de Ven, André L. Delbecq, and Richard Koenig Jr. "Determinants of Coordination Modes within Organizations." In: *American Sociological Review (ASR)* 41.2 (1976), pp. 322–338.

[VHE+07] Giuseppe Valetto, Mary Helander, Kate Ehrlich, Sunita Chulani, Mark Wegman, and Clay Williams. "Using Software Repositories to Investigate Sociotechnical Congruence in Development Projects." In: *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2007, pp. 25–25.

[VR15] Robert Viseur and Gregorio Robles. "First Results About Motivation and Impact of License Changes in Open Source Projects." In: *Proc. IFIP Int. Conf. Open Source Systems: Adoption and Impact*. Springer, 2015, pp. 137–145.

[VVH18] Marat Valiev, Bogdan Vasilescu, and James D. Herbsleb. "Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 644–655.

[VVL17] Michiel Vandecappelle, Nico Vervliet, and Lieven De Lathauwer. "Nonlinear Least Squares Updating of the Canonical Polyadic Decomposition." In: *Proc. Europ. Signal Processing Conf. (EUSIPCO)*. IEEE, 2017, pp. 663–667.

[WB19] Marvin Wyrich and Justus Bogner. "Towards an Autonomous Bot for Automatic Source Code Refactoring." In: *Int. Workshop on Bots in Software Engineering (BotSE)*. IEEE, 2019, pp. 24–28.

[WCE+17] Kangning Wei, Kevin Crowston, U. Yeliz Eseryel, and Robert Heckman. "Roles and Politeness Behavior in Community-Based Free/Libre Open Source Software Development." In: *Information & Management* 54.5 (2017), pp. 573–582.

[WCR+14] Igor S. Wiese, Filipe R. Côgo, Reginaldo Ré, Igor Steinmacher, and Marco A. Gerosa. "Social Metrics Included in Prediction Models on Software Engineering: A Mapping Study." In: *Proc. Int. Conf. Predicitive Models in Software Engineering (PROMISE)*. ACM, 2014, pp. 72–81.

[WD16] Yimin Wei and Weiyang Ding. *Theory and Computation of Tensors: Multi-Dimensional Arrays*. Elsevier, 2016.

[WF94] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[WFW+20] Zhendong Wang, Yang Feng, Yi Wang, James A. Jones, and David Redmiles. "Unveiling Elite Developers' Activities in Open Source Projects." In: *IEEE Transactions on Software Engineering (TSE)* 29.3, 16 (2020).

[WGS03] James Wu, T. C. Nicholas Graham, and Paul W. Smith. "A Study of Collaboration in Software Design." In: *Proc. Int. Sympos. on Empirical Software Engineering* (*ISESE*). IEEE, 2003, pp. 304–313.

[WK03] Laurie Williams and Robert R. Kessler. *Pair Programming Illuminated*. Addison-Wesley, 2003.

[WPA+18] Michel Walrave, Karolien Poels, Marjolijn L. Antheunis, Evert Van den Broeck, and Guda van Noort. "Like or Dislike? Adolescents' Responses to Personalized Social Network Site Advertising." In: *Journal of Marketing Communications* 24.6 (2018), pp. 599–616.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. 2nd ed. Springer, 2012.

[WS20] Mairieli Wessel and Igor Steinmacher. "The Inconvenient Side of Software Bots on Pull Requests." In: *Proc. Int. Conf. Software Engineering Workshops* (*ICSEW*). ACM, 2020, pp. 51–55.

[WS21] Trenton A. Williams and Dean A. Shepherd. "Bounding and Binding: Trajectories of Community-Organization Emergence Following a Major Disruption." In: *Organization Science* 32.3 (2021), pp. 824–855.

[WS98] Duncan J. Watts and Steven H. Strogatz. "Collective Dynamics of 'Small-World' Networks." In: *Nature* 393.6684 (1998), pp. 440–442.

[WSD+09] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. "Predicting Build Failures Using Social Network Analysis on Developer Communication." In: *Proc. Int. Conf. Software Engineering* (*ICSE*). ACM, 2009, pp. 1–11.

[WSS+16] Igor S. Wiese, José T. da Silva, Igor Steinmacher, Christoph Treude, and Marco A. Gerosa. "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*). IEEE, 2016, pp. 345–355.

[WSS+18] Mairieli Wessel, Bruno M. de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana P. Chaves, and Marco A. Gerosa. "The Power of Bots: Characterizing and Understanding Bots in OSS Projects." In: *Proceedings of the ACM on Human-Computer Interaction* (*HCI*) 2.CSCW, 182 (2018).

[WXY+18] Xiao Wang, Lu Xiao, Ye Yang, Xinyi Xu, and Yu Jiang. "Identifying TraIn: A Neglected Form of Socio-Technical Incongruence." In: *Proc. Int. Conf. Software Engineering: Companion Proceedings* (*ICSE Companion*). ACM, 2018, pp. 358–359.

[Wal20] James Walden. "The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL." In: *Proc. Int. Working Conf. Mining Software Repositories* (*MSR*). ACM, 2020, pp. 409–419.

[Wei18] Michael Weiss. "Business of Open Source: A Case Study of Integrating Existing Patterns Through Narratives." In: *Proc. Europ. Conf. Pattern Languages of Programming* (*EuroPLoP*). ACM, 2018, 23.

[XCM06] Jin Xu, Scott Christley, and Gregory Madey. "Application of Social Network Analysis to the Study of Open Source Software." In: *The Economics of Open Source Software Development*. Elsevier, 2006, pp. 247–269.

[XDF16] Qi Xuan, Premkumar T. Devanbu, and Vladimir Filkov. "Converging Work-Talk Patterns in Online Task-Oriented Communities." In: *PLOS ONE* 11.5, e0154324 (2016).

[XF14] Qi Xuan and Vladimir Filkov. "Building It Together: Synchronous Development in OSS." In: *Proc. Int. Conf. Software Engineering* (*ICSE*). ACM, 2014, pp. 222–233.

[XGC+05] Jin Xu, Yongqin Gao, Scott Christley, and Gregory Madey. "A Topological Analysis of the Open Source Software Development Community." In: *Proc. Hawaii Int. Conf. System Sciences* (*HICSS*). IEEE, 2005, 198a.

[XGD+12] Qi Xuan, Mohammed Gharehyazie, Premkumar T. Devanbu, and Vladimir Filkov. "Measuring the Effect of Social Communications on Individual Working Rhythms: A Case Study of Open Source Software." In: *Proc. Int. Conf. Social Informatics* (*SocInfo*). IEEE, 2012, pp. 78–85.

[XJS09] Bo Xu, Donald R. Jones, and Bingjia Shao. "Volunteers' Involvement in Online Community Based Software Development." In: *Information & Management* 46.3 (2009), pp. 151–158.

[YAT16] Zhao Yang, René Algesheimer, and Claudio J. Tessone. "A Comparative Analysis of Community Detection Algorithms on Artificial Networks." In: *Scientific Reports* 6.1, 30750 (2016).

[YCY+22] Likang Yin, Mahasweta Chakraborti, Yibo Yan, Charles Schweik, Seth Frey, and Vladimir Filkov. "Open Source Software Sustainability: Combining Institutional Analysis and Socio-Technical Networks." In: *Proceedings of the ACM on Human-Computer Interaction* (*HCI*) 6.CSCW2, 404 (2022).

[YHH04] Li Yan, Tan Chuan Hoo, and Teo Hock Hai. "The Dynamic Transformation of an Open Source Software Project Leader: A Microorganizational Behavioral Perspective." In: *Proc. Pacific Asia Conf. Information Systems* (*PACIS*). Association for Information Systems, 2004, pp. 2226–2232.

[YK03] Yunwen Ye and Kouichi Kishida. "Toward an Understanding of the Motivation of Open Source Software Developers." In: *Proc. Int. Conf. Software Engineering* (*ICSE*). IEEE, 2003, pp. 419–429.

[YLC15] Yang Yang, Ryan N. Lichtenwalter, and Nitesh V. Chawla. "Evaluating Link Prediction Methods." In: *Knowledge and Information Systems* (*KAIS*) 45.3 (2015), pp. 751–782.

[YMK+15] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E. Hassan, and Naoyasu Ubayashi. "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*). ACM, 2015, pp. 46–55.

[YRS+09] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. “When Private Keys Are Public: Results from the 2008 Debian OpenSSL Vulnerability.” In: *Proc. Conf. Internet Measurement Conference* (*IMC*). ACM, 2009, pp. 15–27.

[YZF23] Likang Yin, Xiyu Zhang, and Vladimir Filkov. “On the Self-Governance and Episodic Changes in Apache Incubator Projects: An Empirical Study.” In: *Proc. Int. Conf. Software Engineering* (*ICSE*). IEEE, 2023, pp. 678–689.

[Yu08] Liguo Yu. “Self-Organization Process in Open-Source Software: An Empirical Study.” In: *Information and Software Technology* (*IST*) 50.5 (2008), pp. 361–374.

[ZC19] Théo Zimmermann and Annalí Casanueva Artís. “Impact of Switching Bug Trackers: A Case Study on a Medium-Sized Open Source Project.” In: *Proc. Int. Conf. Software Maintenance and Evolution* (*ICSME*). IEEE, 2019, pp. 13–23.

[ZCC+19] Weiqiang Zhang, Shing-Chi Cheung, Zhenyu Chen, Yuming Zhou, and Bin Luo. “File-Level Socio-Technical Congruence and its Relationship with Bug Proneness in OSS Projects.” In: *Journal of Systems and Software* (*JSS*) 156 (2019), pp. 21–40.

[ZCM+17] Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. “On the Scalability of Linux Kernel Maintainers’ Work.” In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering* (*ESEC/FSE*). ACM, 2017, pp. 27–37.

[ZG13] Christopher Zachor and Mehmet H. Gunes. “Software Collaboration Networks.” In: *Workshop on Complex Networks* (*CompleNet*). Springer, 2013, pp. 257–264.

[ZLS22] Elijah Zolduoarrati, Sherlock A. Licorish, and Nigel Stanger. “Impact of Individualism and Collectivism Cultural Profiles on the Behaviour of Software Developers: A Study of Stack Overflow.” In: *Journal of Systems and Software* (*JSS*) 192, 111427 (2022).

[ZM10] Minghui Zhou and Audris Mockus. “Developer Fluency: Achieving True Mastery in Software Projects.” In: *Proc. Int. Sympos. on Foundations of Software Engineering* (*FSE*). ACM, 2010, pp. 137–146.

[ZM11] Minghui Zhou and Audris Mockus. “Does the Initial Environment Impact the Future of Developers?” In: *Proc. Int. Conf. Software Engineering* (*ICSE*). ACM, 2011, pp. 271–280.

[ZM12] Minghui Zhou and Audris Mockus. “What Make Long Term Contributors: Willingness and Opportunity in OSS Community.” In: *Proc. Int. Conf. Software Engineering* (*ICSE*). IEEE, 2012, pp. 518–528.

[ZM15] Minghui Zhou and Audris Mockus. “Who Will Stay in the FLOSS Community? Modeling Participant’s Initial Behavior.” In: *IEEE Transactions on Software Engineering* (*TSE*) 41.1 (2015), pp. 82–99.

[ZMM+16] Minghui Zhou, Audris Mockus, Xiujuan Ma, Lu Zhang, and Hong Mei. "Inflow and Retention in OSS Communities with Commercial Involvement: A Case Study of Three Hybrid Projects." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.2, 13 (2016).

[ZND18] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. "Version Control System: A Review." In: *Procedia Computer Science* 135 (2018), pp. 408–415.

[ZSL+22] Yuxia Zhang, Klaas-Jan Stol, Hui Liu, and Minghui Zhou. "Corporate Dominance in Open Source Ecosystems: A Case Study of OpenStack." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 1048–1060.

[ZST+13] Marcelo S. Zanetti, Ingo Scholtes, Claudio J. Tessone, and Frank Schweitzer. "The Rise and Fall of a Central Contributor: Dynamics of Social Organization and Performance in the GENTOO Community." In: *Proc. Int. Workshop Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2013, pp. 49–56.

[ZVK19] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. "What the Fork: A Study of Inefficient and Efficient Forking Practices in Social Coding." In: *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 350–361.

[ZVK20] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. "How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2020, pp. 445–456.

[ZW17] Jianmin Zhang and Xiaojun Wan. "Towards Automatic Construction of News Overview Articles by News Synthesis." In: *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 2017, pp. 2111–2116.

[ZYW11] Wen Zhang, Ye Yang, and Qing Wang. "Network Analysis of OSS Evolution: An Empirical Study on ArgoUML Project." In: *Proc. Int. Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution (IWPSE-EVOL)*. ACM, 2011, pp. 71–80.

[ZZM+19] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. "Companies' Participation in OSS Development – An Empirical Study of OpenStack." In: *IEEE Transactions on Software Engineering (TSE)* 47.10 (2019), pp. 2242–2259.

[ZZW+05] Thomas Zimmermann, Andreas Zeller, Peter Weißgerber, and Stephan Diehl. "Mining Version Histories to Guide Software Changes." In: *IEEE Transactions on Software Engineering (TSE)* 31.6 (2005), pp. 429–445.

[Zhu04] Mu Zhu. *Recall, Precision and Average Precision*. Tech. rep. Waterloo, Canada: Department of Statistics and Actuarial Science, University of Waterloo, 2004, 30.

[Zim20] Théo Zimmermann. "A First Look at an Emerging Model of Community Organizations for the Long-Term Maintenance of Ecosystems' Packages." In: *Proc. Int. Conf. Software Engineering Workshops (ICSEW)*. ACM, 2020, pp. 711–718.