Blekinge Institute of Technology
Doctoral Dissertation Series No. 2025:10
ISSN: 1653-2090
ISBN: 978-91-7295-508-0

Code Review as a Communication Network

Michael Dorner

Doctoral Dissertation

for the degree of Doctor of Philosophy at Blekinge Institute of Technology, to be publicly defended on 2025-09-23 at 14:00 in room J1630, Valhallavägen 1, 37 140 Karlskrona, Sweden

Supervisors	Daniel Mendez, Blekinge Institute of Technology, Sweden Krzysztof Wnuk, Blekinge Institute of Technology, Sweden Darja Šmite, Blekinge Institute of Technology, Sweden
Faculty Opponent	Felix Dobslaw, Mid Sweden University, Sweden
Grading Committee	Emma Söderberg, Lund University, Sweden Brian Fitzgerald, University of Limerick and Lero, Ireland Burak Turhan, University of Oulu, Finland

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2025:10

Code Review as a Communication Network

Michael Dorner

Doctoral Dissertation in Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
Sweden

Blekinge Institute of Technology, Sweden
Department of Software Engineering

Doctoral Dissertation Series No. 2025:10

ISSN: 1653-2090
ISBN: 978-91-7295-508-0
URN: urn:nbn:se:bth-28424

Printed in Sweden, 2025.

To Johann and Anna

Abstract

Background: Modern software systems are often too large and complex for an individual developer to fully oversee, making it difficult to understand the implications of changes. Therefore, most collaborative software projects rely on code review as communication network to foster asynchronous discussions about changes before they are merged. Although prior qualitative studies have revealed that practitioners view code review as a communication network, no formal theory or empirical validation exists. Without formalization and confirmatory evidence, the theory remains uncertain, limiting its credibility, practical relevance, and future development.

Objective: In this thesis, our objective is to (1) formalize the theory of code review as a communication network, (2) empirically evaluate the theory across varied perspectives, contexts, and conditions by quantifying the capability of code review to diffuse information among its participants, (3) demonstrate its practical relevance by applying the theory to the domain of tax compliance in collaborative software engineering, and (4) examine how the role of code review as a communication network for collaborative software engineering may evolve in the future.

Methods: To formalize the theory of code review as a communication network, we developed and validated a simulation model that operationalizes its core propositions about information diffusion among participants. To empirically evaluate the theory, we employed two complementary research approaches. First, we used the simulation model to conduct *in silico* experiments with closed-source code review systems from Microsoft, Spotify, and Trivago, as well as open-source code review systems from Android, Visual Studio Code, and React, to estimate the upper bound of information diffusion in code review. Second, through an observational study, we quantified the diffusion of information in code review across social, organizational, and architectural boundaries at Spotify. To demonstrate the practical relevance of the theory, we analyzed the code review system of a multinational enterprise as a communication network to reveal the latent collaboration structure among developers across borders, which is taxable. To explore the future of code review as a communication network, we conducted a questionnaire survey with 92 practitioners to gather their expectations and discuss how these anticipated changes may reshape our understanding of code review.

Results: By formalizing the theory of code review as a communication network modelled as a time-varying hypergraph, we were able to empirically demonstrate that traditional time-agnostic models substantially overestimate information diffusion in code review. Throughout our empirical studies, we found substential evidence supporting the theory of code review as a communication network: We confirmed that code review is capable of diffusing information quickly and widely among participants, even at a large scale. We also observed extensive information diffusion across social, organizational, and architectural boundaries at Spotify corroborating our theory. However, we also found that information diffusion patterns in open-source code review systems differ significantly, suggesting that findings from open-source environments

may not directly apply to closed-source contexts. Through applying the theory of code review as a communication network in the domain of tax compliance, we were able to uncover the significant and previously unrecognized tax risks associated with collaborative software engineering within multinational enterprises. While practitioners consider code review also in the future a core practice in collaborative software engineering, we identify a potential risk that generative AI may undermine code review's role as a human communication network.

Conclusion: Our work on understanding code review as a communication network contributes not only to theory-driven, empirical software engineering research but also lays the groundwork for practical applications, particularly in the context of tax compliance. Future research is needed to explore the evolving role of code review as a communication network.

Declaration

Publications

The chapters of this compilation thesis are based on six core publications.

Chapter 2

Michael Dorner, Darja Šmite, Daniel Mendez, Krzysztof Wnuk, and Jacek Czerwonka. "*Only Time Will Tell: Modelling Information Diffusion in Code Review with Time-Varying Hypergraphs*". In: *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, September 2022, pp. 195–204. DOI: 10.1145/3544902.3546254

Chapter 3

Michael Dorner, Daniel Mendez, Krzysztof Wnuk, Ehsan Zabardast, and Jacek Czerwonka. "*The Upper Bound of Information Diffusion in Code Review*". In: *Empirical Software Engineering* (June 2023)

Chapter 4

Michael Dorner and Daniel Mendez. "*The Capability of Code Review as a Communication Network*". Manuscript currently under review at *Journal of Empirical Software Engineering*. 2025

Chapter 5

Michael Dorner, Daniel Mendez, Ehsan Zabardast, Nicole Valdez, and Marcin Floryan. "*Measuring Information Diffusion in Code Review at Spotify*". Registered report accepted at *International Symposium on Empirical Software Engineering and Measurement (ESEM)*; manuscript currently under review at *Journal of Empirical Software Engineering*. 2025

Chapter 6

Michael Dorner, Maximilian Capraro, Oliver Treidler, Tom-Eric Kunz, Darja Šmite, Ehsan Zabardast, Daniel Mendez, and Krzysztof Wnuk. "*Taxing Collaborative Software Engineering*". In: *IEEE Software* (2024), pp. 1–8. DOI: 10.1109/MS.2023.3346646

Chapter 7

Michael Dorner, Andreas Bauer, Darja Šmite, Lukas Thode, Daniel Mendez, Ricardo Britto, Stephan Lukasczyk, Ehsan Zabardast, and Michael Kormann. "*Quo Vadis, Code Review?*" Manuscript under review at *IEEE Software*. 2025

Contributions

Michael Dorner is the main author of all six publications compiled in this thesis. As the main author, he took the main responsibility for conceptualization, methodology, software, formal analysis, investigation, data curation, and writing of all chapters. Despite the author taking the main responsibility, the introductory Chapter 1 is written in plural form to emphasize the collaborative nature of all research endeavors. He and the co-authors describe their contributions to the chapters in detail, utilizing the contributor role taxonomy *CRediT*:

Chapter 2

Michael Dorner (Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing: original draft, Visualization, Supervision, Project administration), Daniel Mendez (Conceptualization, Writing: review & editing), Ehsan Zabardast (Conceptualization, Data Curation, Writing: review & editing), Nicole Valdez (Investigation, Data Curation, Writing: review & editing), Marcin Floryan (Resources, Writing: review & editing)

Chapter 3

Michael Dorner (Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing: original draft, Visualization, Supervision, Project administration), Daniel Mendez (Conceptualization, Writing: review & editing), Krzysztof Wnuk (Conceptualization, Writing: review & editing), Jacek Czerwonka (Data Curation)

Chapter 4

Michael Dorner (Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing: original draft, Visualization, Supervision, Project administration), Daniel Mendez (Conceptualization, Writing: review & editing)

Chapter 5

Michael Dorner (Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing: original draft, Visualization, Supervision, Project administration), Daniel Mendez (Conceptualization, Writing: review & editing), Ehsan Zabardast (Conceptualization, Validation, Data Curation, Writing: review & editing), Nicole Valdez (Validation, Investigation, Data Curation, Writing: review & editing), Marcin Floryan (Validation, Writing: review & editing)

Chapter 6

Michael Dorner (Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing: original draft, Visualization, Supervision, Project administration), Maximilian Capraro (Conceptualization, Validation, Writing: review & editing), Oliver Treidler (Conceptualization, Writing: review & editing), Tom-Eric Kunz (Conceptualization, Writing: review & editing), Darja Šmite (Conceptualization, Writing: review & editing), Ehsan Zabardast (Conceptualization, Writing: review & editing), Daniel Mendez (Conceptualization, Writing: review & editing), Krzysztof Wnuk (Conceptualization, Writing: review & editing)

Chapter 7

Michael Dorner (Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing: original draft, Visualization, Supervision, Project administration), Andreas Bauer (Software, Validation, Data Curation, Writing: review & editing, Visualization), Darja Šmite (Conceptualization, Writing: original draft, Writing: review & editing), Lukas Thode (Writing: original draft, Writing: review & editing), Daniel Mendez (Conceptualization, Writing: review & editing), Ricardo Britto (Data Curation, Writing: review & editing), Stephan Lukasczyk (Data Curation, Writing: review & editing), Ehsan Zabardast (Data Curation, Writing: review & editing), Michael Kormann (Validation, Writing: review & editing)

Funding

This research was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

Open Science

To enhance transparency, reproducibility, and future research, we adhere to the open science principle to the best extent possible through making publications freely available online without access barriers (*open access*), ensuring data is openly accessible, reusable, editable, and shareable for any research purpose (*open data*), and releasing research software under licenses that allow anyone to use, modify, and distribute it—modified or not—to everyone free of charge (*open source*). We have made all data (including anonymized raw and the processed datasets), software (including documentation, and supplementary materials) from this dissertation publicly available, except for the data used in Chapter 6, which remains strictly confidential due to the sensitive nature of tax compliance.

We also ensured strict anonymization of all human participants and, wherever requested, upheld confidentiality to our industry partners to comply with legal and ethical standards for handling research data. Through our commitment to open science, we aim to promote validation of our findings, encourage extensions of our work, and contribute meaningfully to the broader scientific community.

Table 1 presents an overview of our open science efforts associated with each chapter and study in this dissertation.

Table 1: Overview of our open science efforts for each chapter of this dissertation

Chapter	Study	Open Access	Open Data	Open Source
Chapter 2	[41]	yes	yes	yes
Chapter 3	[39]	yes	yes	yes
Chapter 4	[38]	yes	yes	yes
Chapter 5	[40]	yes	yes	yes
Chapter 6	[37]	yes	no	yes
Chapter 7	[36]	yes	yes	yes

Acknowledgements

This thesis would not have been possible without the support and contributions of many extraordinary people, to whom I am deeply grateful.

First and foremost, I would like to express my deepest gratitude to my supervisors, Daniel Mendez, Krzysztof Wnuk, and Darja Šmite, who have been excellent mentors throughout this journey. Their exceptional guidance, unwavering belief in my abilities and ideas, and everlasting patience have been indispensable throughout every step of my research. I truly learned from the best.

This research would not have been possible without the contributions, support, shared laughter, and feedback from Maximilian Capraro, who was the dependable partner from the very beginning of my academic journey as supervisor of my master thesis to the two research spin-offs we founded, Julian Frattini, my long-term office mate and flatmate whose critical feedback was always outstanding and helpful, Andreas Bauer, who never got tired of helping out and is the software engineer I strive to be, and last but not least, Ehsan Zabardast, who was my guy for everything anytime anywhere. I am even more grateful to know those people will continue to be part of my future journeys in one way or another.

I am deeply indebted to all of my co-authors, in particular to Oliver Treidler and Tom-Eric Kunz, who made our advances in the interdisciplinary research on taxation in software engineering through their expertise in taxation and beyond possible.

My sincere appreciation also goes to the participants of the studies and my industry partners—Microsoft, Spotify, Trivago, SAP, Ericsson, JetBrains, and a Nordic bank—without whom this research would not have been possible. In particular, I am deeply indebted to Jacek Czerwonka, Nicole Valdez, Floryan Marcin, Andy Grunwald, Simon Bruegge, Riccardo Britto, Stephan Lukasczyk, and Michael Kormann for their support, trust, and manifold contributions to my research.

I am immensely thankful to the Department of Software Engineering at Blekinge Institute of Technology, led by Michael Mattsson, and to the SERT Research Project, led by Tony Gorschek, for providing this extraordinary academic environment I had the pleasure to be part of. I am particularly grateful to my friends and colleagues at the department for their camaraderie, stimulating discussions, and manifold support. In particular, I would like to thank Waleed Abdeen, Florian Angermeier, Parisa Elahidoost, Anna Eriksson, Davide Fucci, Felix Jedrzejewski, Anders Sundelin, and Lukas Thode for making the department so much more than a workplace for me.

Last but not least, I extend my heartfelt thanks to my wonderful wife Christina for her unconditional love, unwavering belief in my abilities, and continuous encouragement throughout this journey. Her support and understanding have made all the difference, and I could not have done it without her.

Thank you all!

Contents

1

Introduction

1

1.1 Motivation

During World War II, the remote islands in the Pacific became strategic military bases for Japanese and Allied forces. The sudden arrival of soldiers, aircraft, and supplies had a profound impact on the indigenous populations, who were until then largely isolated from the modern world. These native islanders, witnessing unprecedented technological advancements and the arrival of valuable goods, struggled to comprehend the significance of the events unfolding before them.

As the war concluded and the military presence diminished, some island communities developed what anthropologists later termed *cargo cults*. In an attempt to replicate the circumstances that led to the arrival of the valuable cargo, these communities began constructing makeshift runways, control towers, and model airplanes. They mimicked the behaviors of the departing military personnel, engaging in rituals they believed would attract the return of the supplies.

The cargo cults represented a form of imitation rooted in a lack of understanding. The islanders observed the military activities but lacked insight into the underlying technological, logistical, and strategic reasons for those activities. Consequently, their attempts to replicate the rituals were superficial and devoid of the necessary context.

Fast forward, to the realm of software engineering. During the early days of software engineering, code was often designed and implemented by individual developers working in isolation. As the software systems under development became more and more complex, software engineering became a collaborative effort requiring many developers with diverse expertise and specializations, organized in different teams, and later even distributed around the globe. To maintain and enhance a collective understanding of the codebase and the changes to it, developers began to discuss a code change before it gets merged into the codebase. Those discussions had different forms: as a waterfall-like procedure in formal, heavyweight code inspections in the 1980s [45] or as synchronous and steady exchange between two developers as practiced in pair programming [127]. Nowadays, the asynchronous and less formal discussions around a code change are referred to as *code reviews*. To facilitate these discussions, companies and open-source projects use multipurpose communication platforms like mailing lists in the Linux kernel [130], internally developed tools such as CodeFlow at Microsoft [39] and Critique at Google [106], open-source tools like Phabricator at Facebook [79] and Gerrit for Android and Chromium, or platforms supporting pull/merge requests like GitHub and GitLab [39]. Although the tools may differ in implementation details, they all share the common goal of facilitating a communication network for developers to discuss code changes. These discussions have become an integral part of the daily routine of software engineers [17, 36].

But how confident are we that code review actually works as intended? Are our current implicit assumptions about code review as a communication network robust

enough to justify its widespread adoption and continual refinement in both industry and academia? Can we rely on this implicit theory for practical applications in real-world setting? How certain are we that code review has not become a cargo cult of collaborative software engineering, a ritual repeated out of habit, convention, or social expectation, rather than grounded in a clear, evidence-based, and comprehensive understanding?

1.2 Research Objective & Questions

In this thesis, we set out to fill this gap: Our objective is to (1) formalize the theory of code review as a communication network, (2) empirically evaluate the theory across varied perspectives, contexts, and conditions by quantifying the capability of code review to diffuse information among its participants, (3) demonstrate its practical relevance by applying the theory to the domain of tax compliance in collaborative software engineering, and (4) examine how the role of code review as a communication network for collaborative software engineering may evolve in the future.

The following subsection introduces each objective, explains its motivation, and presents the corresponding research questions in detail.

1.2.1 Theory Formalization

The theory of code review as a communication network builds not only on the argument presented in the motivation on how code review is used in collaborative software engineering, but also on a body of exploratory studies that investigate the motivations for and expectations toward code review in industrial settings [7, 14, 17, 29, 106]. Our synthesis of this prior research revealed a consistent pattern: practitioners seem to perceive the information[1] exchange through code review as the common cause behind the observed and desired effects of code review, which is the foundation of our understanding of code review as a communication network [39]. However, this theory—as often in software engineering research [115]—remains implicit and has yet to be formalized.

Therefore, our first objective is to formalize the theory through the capacity of code review to bring information from where it is available to where it is needed. We term this process *information diffusion*. In physics, diffusion refers to the movement of

[1]Throughout this work, we intentionally use the term *information* rather than *knowledge*, following the reasoning of [39] and [92], even though earlier work used the latter term [7, 14, 17, 29, 106]. Although not synonymous, information can be seen as a superset of knowledge: knowledge is meaning derived from information through interpretation. Not all information constitutes knowledge, but all knowledge originates as information. This framing allows us to subsume differing notions of knowledge without engaging in epistemological debates or addressing truth claims often associated with knowledge. Code review may include subjective elements such as opinions, assumptions, or misunderstandings; forms of information that may not meet all definitions of knowledge or truth.

particles from regions of higher concentration to regions of lower concentration, leading to a gradual equalization over time. By analogy, information diffusion describes the spread of information through the channels a communication network provides. Through a set of explicit propositions about the information diffusion capabilities of code review, this formalization makes the theory's underlying assumptions, mechanisms, and expectations transparent and empirically assessable, enabling a deeper understanding of code review beyond surface-level practices.

Given the fine granularity, high complexity, and scale of human communication in code review, this formalization requires a suitable abstraction from reality, a model of code review as a communication network that captures how information diffuses. This leads us to our first research question (RQ_0): What model best represents information diffusion in code review?

1.2.2 Theory Evaluation

Although the prior qualitative, exploratory work [7, 14, 17, 29, 106] provided a solid foundation for the theory of code review as a communication network, its formalization based on prior work alone is not sufficient to evaluate the theory's validity or applicability in practice.

Exploratory research begins with specific observations, distills patterns in those observations, typically in the form of hypotheses, and derives theories from the observed patterns through (inductive) reasoning. The nature of exploratory and especially qualitative research enables analysis of chosen cases and their contexts in great detail. However, their level of evidence is also limited as they are drawn from those specific cases (especially when there is little to no relation to existing evidence that would allow for generalization by analogy). Therefore, while such initial theories may well serve as a very valuable foundation, they still require multiple confirmatory evaluation" as, otherwise, their robustness (and scope) remains uncertain rendering it difficult to establish credibility, apply in practice, or develop it further. We thus postulate that exploratory (inductive) research alone is not sufficient to achieve a strong level of evidence, as discussed also in greater detail by Wohlin [128].

Confirmatory research, in turn, typically forms a set of predictions based on a theory (often in the form of hypotheses) and validates to which extent those predictions hold true or not against empirical observations (thus evaluating the consequences of a theory). To efficiently build a robust body of knowledge, we need both exploratory and confirmatory research to minimize bias and maximize the validity and reliability of our theories efficiently. Figure 1.1 shows this interdependency between exploratory (theory-generating) and confirmatory (theory-evaluation) research in empirical research.

Therefore, our second objective is to evaluate the formalized theory of code review as a communication network in different empirical, confirmatory studies.

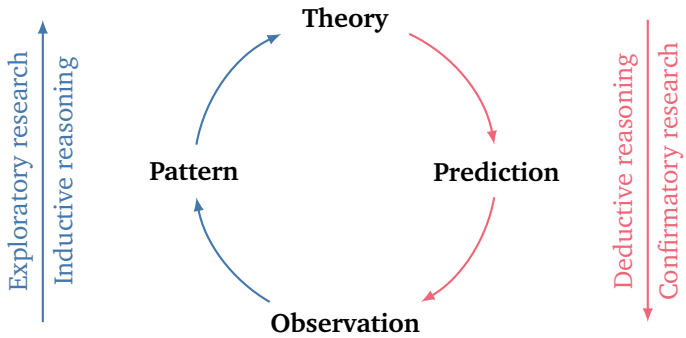


Figure 1.1: The empirical research cycle (in analogy to [77]): While exploratory research is theory-generating using inductive reasoning (starting with observations), confirmatory research is theory-evaluation using deductive reasoning (starting with a theory).

Since confirmatory research can only invalidate hypotheses (and falsify theories), we aim to find evidence that could not be aligned with the formalized theory of code review as a communication network or its universality; further constraints, context, or limitations may must be considered. The inversion of an argument is not valid: if we cannot falsify the theory, it is not necessarily true. Yet, it provides evidence to our current understanding and removes it from the threat of code review as cargo cult.

In the absence of a predefined threshold for when to reject our hypotheses, we further adopt a more data-driven interpretation grounded in patterns observed across our empirical studies. That is, rather than establishing an arbitrary cutoff *a priori*, we discuss information diffusion within our sample of code review systems to identify consistent deviations from the expectations encoded in the propositions. Accordingly, we use the term *theory evaluation* rather than *theory testing* throughout this dissertation to avoid suggesting that our approach is based on statistical hypothesis testing.

To provide the empirical foundation for this comprehensive discussion, we formulate the following four research questions: How widely (RQ_1) and how quickly (RQ_2) can information spread in closed-source code review and widely (RQ_3) and how quickly (RQ_4) can information spread in open-source code review?

1.2.3 Theory Application

To demonstrate the practical relevance of our theory, we apply code review as a communication network in the emerging field of tax compliance in collaborative software engineering. By applying our theory, we aim to uncover previously unrecognized cross-border collaboration among software engineers by analyzing the code review system of a multinational enterprise as a communication network to reveal the latent collaboration structure among developers across borders because this cross-border

collaboration within a multinational enterprise has a critical, yet often overlooked, legal implication: cross-border collaboration is taxable [37]. The ongoing litigation between Microsoft and the U.S. Internal Revenue Service (IRS), in which the IRS claims Microsoft owes an additional $28.9 billion in taxes, along with penalties and interest [124], underscores the urgency and the importance of understanding and addressing these risks. By applying the theory of code review as a communication network in the context of tax compliance, we contribute to a broader understanding of how collaboration among developers intersects with legal and organizational accountability in collaborative and distributed software engineering.

1.2.4 Theory Evolution

In Chapter 7, we explore what changes developers anticipate over the next five years. By surveying practitioners from multiple software-driven companies across diverse domains, the study aims to capture expectations about the future of code review, especially in light of emerging technologies like large language models (LLMs), and to reflect on their implications for code review as a communication network in collaborative software engineering.

1.3 Research Design

This section outlines the overall research design of the dissertation. Building on the four research objectives (i.e., the formalization, evaluation, application, and evolution of the theory of code review as a communication network) and ten associated research questions (RQ_0 to RQ_9) introduced in the previous section, this section describes how each is addressed through empirical studies. This provides the structural link between the dissertation's theoretical foundations and its empirical contributions.

To provide a high-level overview, Table 1.1 summarizes how each chapter contributes to the dissertation's objectives. It maps chapters to their corresponding research objective, research questions, methods, and data sources, offering the reader a complete overview of the research design.

The remainder of this section explains each component of the research design in more detail. Section 1.3.1 outlines how the research objectives and research questions map to the chapters. Section 1.3.2 discusses the research methods employed, and Section 1.3.3 summarizes the data sources used across the different studies.

1.3.1 From Theory to Empirical Studies

This subsection outlines the research path from the formalization of the theory of code review as a communication network to its empirical evaluation, practical application, and future exploration.

Chapter	Research Objective	Research Question	Research Method	Research Data
Chapter 2	Theory Formalization	What model best represents information diffusion in code review? (RQ_0)	*in silico* experiment	All code reviews at Microsoft over four weeks
Chapter 3	Theory Formalization & Evaluation	How widely (RQ_1) and how quickly (RQ_2) can information spread in closed-source code review?	*in silico* experiment	All code reviews at Microsoft, Spotify, and Trivago over four weeks
Chapter 4	Theory Formalization & Evaluation	How widely (RQ_3) and how quickly (RQ_4) can information spread in open-source code review?	*in silico* experiment replication	All code reviews of four weeks at Android, React, and Visual Studio Code, along with those from Chapter 3
Chapter 5	Theory Formalization & Evaluation	What is the extent to which information diffuses across social (RQ_5), organizational (RQ_6), and software architectural (RQ_7) boundaries?	Observational study	All code reviews, organizational data, and software architectural description of Spotify over one year
Chapter 6	Theory Application	What is the extent of cross-border code reviews in a multinational company? (RQ_8)	Observational study	All code reviews at an anonymous multinational enterprise over five years
Chapter 7	Theory Evolution	What changes in code review do practitioners foresee in the future? (RQ_9)	Questionnaire survey	Responses from ca. 92 practitioners from SAP, Ericsson, JetBrains, and a large Nordic bank

Table 1.1: Overview of chapters, research objectives, research questions, research methods, and research data used in this dissertation.

1

First, we begin with the formalization of a theory of code review as a communication network. The theory is articulated through a set of propositions about how information diffuses during code review. These propositions were developed iteratively and operationalized across multiple studies, as presented in Chapters 2 to 5. Although the formalized theory is presented in Section 1.4, we additionally include it here to enhance readability, support understanding, and provide a foundation for its evaluation, application, and exploration.

Code Review as a Communication Network

Our research is grounded in the theory that code review functions as a communication network which has the capacity to spread information among its participants. We refer to this phenomenon as *information diffusion*.

From this perspective, we propose the following theory encoded in seven propositions: If code review is a functional communication network, then (a) code review is capable of spreading information widely (P_1) and quickly (P_2) in closed-source and widely (P_3) and quickly (P_4) in open-source collaborative software engineering, and (b) information diffuses in code review

- through social boundaries between developers (P_5) and
- through organizational boundaries between teams (P_6) and
- through architectural boundaries between software components (P_7)

We can formulate the theory using our hypotheses as the propositional statement

$$T \implies (P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5 \wedge P_6 \wedge P_7) \tag{1.1}$$

where T denotes the theory that code review functions as a communication network.

The foundation for this formalization is laid in Chapter 2, where we develop and validate a simulation model based on time-varying hypergraphs. This model serves as a central abstraction for studying information diffusion and addresses our first research question: What model best represents information diffusion in code review (RQ_0)?

Second, the theory is evaluated through three empirical studies. These studies examine whether the expectations expressed in seven propositions are supported by evidence:

- that it spreads widely (P_1) and quickly (P_2) in closed-source settings, and widely (P_3) and quickly (P_4) in open-source settings; and
- that information diffuses across social (P_5), organizational (P_6), and architectural (P_7) boundaries.

Given the formalization of the theory as a propositional statement where the theory implies the conjunction of all seven propositions, it can only be corroborated in its current form if all of the propositions are supported by empirical evidence. To enable empirical evaluation, the propositions are operationalized into observable constructs. Propositions P_1 to P_4 are operationalized through a simulation model introduced and validated in Chapter 2. Propositions P_5 to P_7 are addressed directly in Chapter 5 where they are studied. The following chapters provide the empirical foundation for evaluating each of the theory's propositions:

- Chapter 3 investigates how widely (P_1) and how quickly (P_2) information can spread in the closed-source code review systems at Microsoft, Spotify, and Trivago, addressing RQ_1 and RQ_2.
- Chapter 4 investigates how widely (P_3) and how quickly (P_4) information can spread in the open-source code review systems at Android, React, and Visual Studio Code, addressing RQ_3 and RQ_4.
- Chapter 5 investigates the information diffusion across social, organizational, and architectural boundaries P_5 – P_7 in the code review system at Spotify, addressing RQ_5, RQ_6, and RQ_7.

The annotated formula below summarizes how the theory formalized as a propositional statement is mapped to the chapters in which each proposition is operationalized and evaluated.

$$T \implies \Big(P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5 \wedge P_6 \wedge P_7 \Big) \tag{1.2}$$

Evaluated in Chapter 3

Evaluated in Chapter 4

Operationalized and evaluated in Chapter 5

Operationalized in Chapter 2

Third, we apply the theory to the domain of international tax compliance. This study uses the theory to identify cross-border collaboration patterns in code review that may have implications for tax compliance of multinational enterprises. To this end, we formulate the following research question: What is the extent of cross-border code reviews in a multinational company (RQ_8)? Chapter 6 addresses RQ_8 by modelling code review at a large multinational enterprise as a communication network.

Fourth and last but not least, the theory's future relevance is explored through a survey of practitioners across multiple software-driven organizations. This study examines how developers expect code review practices to change over time. Chapter 7 explores

the future of code review as a communication network by asking RQ_9: What changes in code review do practitioners foresee in the future?

1.3.2 Research Methods

In this research, we conducted a variety of experimental (interventional) and observational (non-interventional) studies to address all ten research questions. Specifically, we employed *in silico* experiments and observational studies to evaluate and apply our theory, and a questionnaire survey to explore its future.

We justify and describe all research methods in detail in each respective chapter where they were applied. However, given the rarity of simulation-based studies in software engineering research, along with the often unclear distinction between experiments and observational studies in the field [6] and the ambiguity of the term "survey" [114], we discuss our three research methods in the following subsections.

Experimental Studies

Experimental studies are studies where an intervention is deliberately introduced to observe its effects on some aspects of the reality under controlled, modelled conditions [6]. Thus, all experiments imply an inherent trade-off in the design and interpretation of experimental studies between realism and control. Depending on which parts of the experiment (setting or subject) are controlled and modelled by the researcher, we know four types of experiments: *in vivo* (natural subjects and natural settings), *in vitro* (natural subjects and modelled settings), *in virtuo* (natural subjects and settings modelled as computer model), and *in silico* (both subjects and settings are modelled as computer model by the researcher) experiment [63]. Table 1.2 summarizes those four types.

Conducting interventional studies in the context of code review presents significant challenges. On one hand, code review lies at the heart of collaborative software engineering. Organizations are understandably hesitant to risk disruptions to these essential processes or jeopardize costly and important operations like code review. This makes *in vivo* experiments in a natural setting infeasible.

On the other hand, modelling a controlled environment that accurately reflects the complex nature of software engineering in which code review takes place is equally challenging. Code review is deeply interwoven with the engineering of complex software system and involves a variety of tools, practices, and team dynamics, making it difficult to isolate variables without oversimplifying the setting. Moreover, the scope of code review as a practice often extends beyond groups of developers, encompassing entire organizations or communities, sometimes involving thousands of developers. This interdependency with collaborative software engineering and the sheer size and

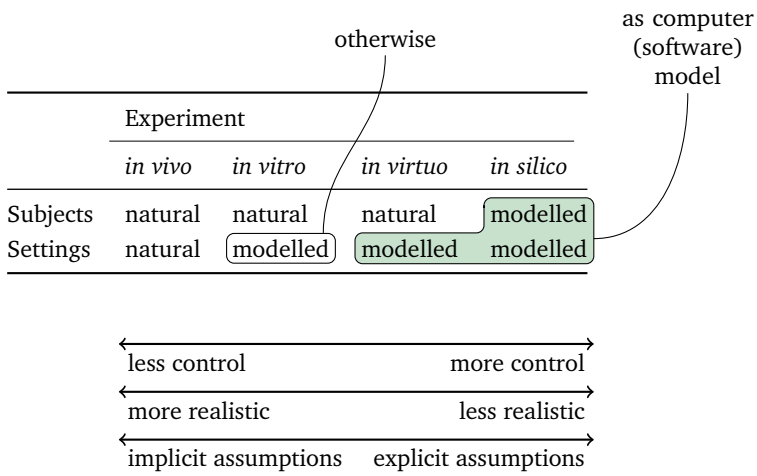


Table 1.2: A comparison of *in vivo*, *in vitro*, *in virtuo*, and *in silico* experiments with respect to subjects and settings.

pace of code review further complicate efforts to model or experiment with code review effectively.

In our work, we tackle these two challenges for experimental studies by employing *in silico* experiments (or simulations). While still relatively uncommon in software engineering [39], *in silico* experiments have been the cornerstone of significant advancements in other disciplines. For example, traditional experiments in climate research are not feasible because the Earth's climate system is vast, complex, and interconnected, making it impossible to manipulate or isolate specific variables without causing widespread and potentially irreversible impacts. Additionally, the scale and timescales involved are far beyond what can be controlled or observed in a traditional experimental setting. Yet, for example, Klaus Hasselmann and Syukuro Manabe achieved groundbreaking results using computer models to experiment with the Earth's climate system, which lead to the 2021 Nobel Prize in Physics "for the physical modelling of Earth's climate, quantifying variability, and reliably predicting global warming."

We use *in silico* experiments for two purposes:

1. To validate our simulation model and answer research question RQ_0 in Chapter 2 which model best represents information diffusion in code review.
2. To evaluate propositions P_1 to P_4 and address the research questions on how widely and how quickly information can spread in closed-source (RQ_1, RQ_2) and open-source (RQ_3, RQ_4) code review systems, as discussed in Chapters 3 and 4.

Observational Studies

In contrast to experimental studies, an *observational study* (or *field study*) refers to any research conducted in a specific, real-world setting to study a specific software engineering phenomenon [114]. Observational studies are unobtrusive in that a researcher does not actively control or change any parameters or variables. That is, there is no deliberate modification of the research setting. Observational studies are used to develop a deep understanding of phenomena in specific, concrete, and realistic settings—the specific setting may refer (among others) to a particular software system, organization, or team of individuals. For this reason, an observational study offers maximum potential for capturing a realistic context, unlike, for example, an interventional study [114]. However, this realism is gained at the price of a low precision of measurement of behavior and a low generalizability of findings [114].

In Chapter 5, we present an observational study of code review at Spotify to examine the extent to which information diffuses (a) between code review participants (proposition P_1), (b) across software components (proposition P_2), and (c) between teams (proposition P_3). This allows us to understand the phenomenon information diffusion

in code review at specific, concrete, and realistic setting at Spotify. As observational study, the precision of our measurements regarding the behavior of code review participants is inherently limited and we cannot claim any generalizability to other settings (i.e., companies or open-source software projects and their code review systems). However, following the deductive logic of confirmatory research, observing even a single setting where code review does not function as a communication network would already invalidate our theory as it stands today and requires refining the theory.

Aside from the purpose of theory validation, we also report our observational study to measure cross-border code reviews—instances where participants are employed by legal entities in different countries—at a large multinational enterprises (addressing research question RQ_8) as a proxy for collaborative software engineering across international borders in the context of tax compliance in Chapter 6 to demonstrate the theory's practical relevance.

Questionnaire Survey

A survey is a research method used to collect data from a predefined group of respondents in order to gain information and insights on various topics of interest. Although a survey is unobtrusive—meaning the researcher does not manipulate any variables during data collection, similar to an observational study—it is designed to maximize generalizability across a population [114]. However, the extent to which the findings can be generalized is influenced by the sampling strategy, as the representativeness of the sample directly impacts the validity and applicability of the results.

The term survey is considered overloaded [114]. In this research, we understand surveys as a research method conducted through questionnaires or interviews, aimed at collecting quantitative or qualitative data on attitudes, behaviors, experiences, demographics, or opinions. To avoid ambiguity, we henceforth use the term *questionnaire survey* for clarity.

In Chapter 7, we conducted a questionnaire survey of 92 practitioners from SAP, Ericsson, JetBrains, and a large Nordic bank to investigate the changes they anticipate in the future of code review and answer our last research question RQ_9. We employed *quota sampling*, a non-probalistic, multistage sampling approach [9]. Given the absence of a comprehensive sampling frame for all developers, we approximated one by dividing the target population into sub-frames, specifically, individual companies, with each contributing approximately 25 professional developers. These participants were selected in a non-random but roughly equal manner across companies.

Chapter	Research Data
Chapter 2	All code reviews at Microsoft over four weeks
Chapter 3	All code reviews at Microsoft, Spotify, and Trivago over four weeks
Chapter 4	All code reviews at Android, React, and Visual Studio Code over four weeks, along with those from Chapter 3
Chapter 5	All code reviews, organizational data, and software architectural descriptions from Spotify over one year
Chapter 6	All code reviews at an anonymous multinational enterprise over five years
Chapter 7	Responses from approximately 92 practitioners at SAP, Ericsson, JetBrains, and a large Nordic bank

Table 1.3: Overview of chapters and research data used in this thesis.

1.3.3 Research Data

The empirical studies presented in this dissertation are based on multiple datasets collected from both industrial and open-source software development contexts. These datasets encompass closed-source and open-source code review systems, as well as survey data from professional practitioners. Each source was selected to align with specific research questions, ensuring that the resulting analyses are grounded in relevant, representative, and appropriately scaled data.

Table 1.3 provides an overview of the datasets used in each chapter; a detailed description of each dataset and its collection can be found in the respective chapter.

Code Review Data

We collected code review data from six large-scale code review systems: three closed-source systems provided by collaborating companies (Microsoft, Spotify, and Trivago) and three open-source systems (Android, Visual Studio Code, and React), addressing research questions RQ_0–RQ_7 as outlined in Chapters 2 to 4.

Although industrial and open-source code review share many structural features, they differ in important ways:

- Closed-source code review systems are bounded to an organization, contributors are typically employees, and review responsibilities are distributed across formal teams.
- Open-source code review systems are often led by companies but open to external contributors. They usually centralize review activity around a smaller set of highly active maintainers.

Table 1.4: Code review systems used to parameterize the two *in silico* experiments. All datasets are publicly available.

Code review system	Size	Code reviews	Participants	Tooling
Open source				
Android	large	10 279	1793	Gerrit
Visual Studio Code	mid-sized	802	162	GitHub
React	small	229	64	GitHub
Closed source				
Microsoft	large	309 740	37 103	CodeFlow
Spotify	mid-sized	22 504	1730	GitHub
Trivago	small	2442	364	BitBucket

Across both categories, systems were chosen to ensure diversity in size and code review tooling to avoid bias. Table 1.4 provides a detailed overview of the code review data used in this thesis.

In addition to supporting our *in silico* experiments, we collected a separate closed-source code review system to measure cross-border collaboration within a large multinational enterprise that requested anonymity, thereby addressing RQ_8 in Chapter 6. This dataset is the only one not publicly available, for understandable reasons.

Survey Data

To complement the code review data, we conducted a questionnaire survey with professional developers, as described in Chapter 7. The survey addressed research question RQ_9 and gathered both quantitative and qualitative information on code review practices, attitudes, and perceived challenges, helping to contextualize and interpret our empirical findings. All responses were anonymized, and the study was conducted in accordance with ethics guidelines.

1.4 Results

We proposed and formalized the theory that code review functions as a communication network through its capability to diffuse information among its participants. To make this theory empirically assessable, we define it as a propositional statement composed of seven concrete, testable propositions: If code review is a functional communication network, then (a) code review is capable of spreading information widely (P_1) and quickly (P_2) in closed-source and widely (P_3) and quickly (P_4) in open-source collaborative software engineering, and (b) information diffuses in code review

- through social boundaries between developers (P_5) and
- through organizational boundaries between teams (P_6) and
- through architectural boundaries between software components (P_7)

We can formulate the theory using our hypotheses as the propositional statement

$$T \implies (P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5 \wedge P_6 \wedge P_7) \tag{1.3}$$

where T denotes the theory that code review functions as a communication network. Together, these propositions capture the breadth, pace, and boundary-crossing nature of information diffusion through code review. For further details, see Section 1.3.1 earlier and the empirical studies presented in Chapters 2 to 4.

To formalize and operationalize this theory, we developed and validated a simulation model as an abstraction of reality based on time-varying hypergraphs that enables the empirical evaluation of information diffusion in code review through *in silico* experiments. The validation of this simulation model resulted in two key contributions. First, we established that time-agnostic models substantially overestimate the extent of potential information diffusion. In contrast, our simulation model, grounded in time-varying hypergraphs, offers a rigorous mathematical and theoretical framework for accurately capturing and analyzing the dynamics of information diffusion in code review at Microsoft. Second, through the validation of the conceptual model, we accidentially introduced a novel generalization of Dijkstra's algorithm for computing minimal paths in time-varying hypergraphs. While not directly situated within software engineering, this contribution may be seen as advancing the theoretical foundations of computer science. Details of the simulation model, its validation, and the generalized Dijkstra's algorithm for time-varying hypergraphs are presented in Chapter 2. To ensure the correctness of this novel and previously unimplemented algorithm, we extensively validated the computer model that implements the conceptual model in software. For this purpose, we developed a comprehensive test suite in close collaboration with students from a software testing course at Blekinge Institute of Technology. Our experiences from this collaboration are reported in a dedicated study [35], which is not part of this thesis.

Throughout our empirical studies, we found substantial evidence supporting the theory of code review as a communication network.

Through our two *in silico* experiments presented in Chapters 3 and 4, we found that code review networks are capable of spreading information widely and quickly. However, this capability is not equally distributed. Our findings reveal substantial differences not only among the individual systems, but also between open-source and closed-source code review systems. While open-source projects tend to spread information more quickly, they do so across a significantly smaller fraction of participants

compared to their closed-source counterparts. This suggests that findings derived from studies of open-source code review may not necessarily generalize to closed-source environments. Given these structural and behavioral differences, we advocate for a critical reevaluation of the generalizations made in the software engineering research community based primarily on open-source data.

Our analysis of code review at Spotify reveals that information diffusion occurs across social, organizational, and architectural boundaries. Socially, code reviews are highly distributed: Over 99% of review pairs involve entirely distinct sets of participants, indicating minimal overlap and suggesting broad diffusion across developer groups. Organizationally, while 81% of reviews involve developers from a single team, approximately 18% span multiple distinct teams demonstrating nontrivial cross-team interaction. Architecturally, diffusion is also evident as code reviews almost always link across multiple repositories, reflecting communication and collaboration across different software components. These findings collectively support the notion that code review at Spotify functions as a communication network facilitating widespread information diffusion—corroborating our theory of code review as a communication network. A detailed analysis is presented in Chapter 5.

By applying our theory in the context tax compliance, we uncovered the significant tax risk associated with collaborative software engineering within multinational enterprises. Through measuring cross-border code reviews, i.e. code reviews with participants from different countries, over the timespan of four years at a large, multinational enterprise to approximate the collaborative software engineering. We found that the share of cross-border code reviews was between 6% and 10% in 2019 and 2020. Yet, we see a further steep increase reaching between 25% and 30% at the end of 2022. Interestingly, 6% of all cross-border code reviews involve participants from more than two countries. This means tax compliance pricing in collaborative software engineering becomes not only a bilateral but a multilateral problem with not only two but multiple—in our case company up to six—different jurisdictions and tax authorities involved. Although the share of cross-border collaboration may vary among companies, yet, our findings suggest that—through the proxy of cross-border code reviews—cross-border collaboration becomes or is already a significant part of daily life in multinational software companies. Further details can be found in Chapter 6.

The role of code review might evolve: According to our survey, practitioners expect code review to be a fundamental aspect of software engineering in the future: 71% of all respondents expect to spend about the same amount or more time on code review and to review more artifacts in the future. However, the vast majority also expects generative AI to become a major participant in code review. Taking a perspective of skeptics, we reflect three challenges that derive from the trends captured by our survey, that may ultimately signal the end of code review as we know it: erosion of understanding, erosion of accountability, and erosion of trust. We advocate for further

research to explore the implications of integrating LLMs and other AI technologies into code review, as we believe code review is where the effects of LLM adoption in collaborative software engineering are likely to surface first. An over-reliance on AI subtly shifts the locus of understanding from human teams to the LLM—an entity that, despite its capabilities, cannot be held accountable, reason contextually, or justify its decisions beyond probabilistic associations. To preserve the integrity of software development as collaborative, transparent, trustworthy, and responsible practice, we must ensure that code review remains grounded in human understanding and oversight. Further details can be found in Chapter 7.

1.5 Discussion

While detailed discussions of the results and limitations of each individual study are provided in the respective chapters, this section critically reflects on the overarching results, contributions, and limitations of this cumulative dissertation, and considers their implications for the theory of code review as a communication network.

By formalizing the theory of code review as a communication network, we make it explicit, facilitating effective communication across research and practice. This provides a shared language that unifies efforts to enhance code review as a core practice in collaborative software engineering. Although we were able to integrate

However, we missed the opportunity to formalize the theory early on and instead followed a research-then-theory approach [115]. Although the introduction and its structure might suggest that we followed a systematic approach such as that proposed by [109], the alignment with key elements of their framework (i.e., constructs, propositions, and empirical grounding) emerged retrospectively rather than being applied systematically from the outset. This is also reflected in the fact that the theory became more explicit only over time.

The empirical evaluation of the theory provides a deeper understanding of code review, revealing that it functions as a communication network, diffusing information across social, organizational, and architectural boundaries, and showing its potential capacity to diffuse. This adds a new perspective to the conventional view of code review as a simple review or quality assurance mechanism and emphasizes its vital role in facilitating information exchange across diverse teams, departments, and components. Software projects and organizations may benefit from treating code review as part of their broader communication and coordination infrastructure for their software engineering efforts. Adopting this perspective, as substantiated by our research, lays the foundation for more effective code review tools and their integration, reduces the risk of silos within companies, fosters a shared understanding of the codebase, and enables earlier detection of communication bottlenecks.

The validation of our theory relies heavily on simulations as an empirical method. In our case, traditional experimental methods were not feasible—no company would allow us to run an experiment on their entire code review system for several weeks to answer specific research questions. Although *in silico* experiments are still uncommon in software engineering research and challenged to be an empirical research method [114], they proved to be an effective tool in this research. By creating controlled, reproducible research environments, *in silico* experiments enabled us to evaluate the propositions and explore phenomena that would have been difficult, if not impossible, to examine in real-world settings. Our research demonstrates how simulations can significantly enrich the empirical research toolkit in software engineering, especially when traditional methods are impractical or infeasible.

Simulations with their explicit modelling and parametrization are particularly powerful in the context of open science. Therefore, we adhere to the open science principle by making all data (including anonymized raw and processed datasets), software (including documentation), and supplementary materials publicly available [76]. However, the efforts required to thoroughly anonymize large-scale and diverse datasets are substantial, and the data release process at companies is often cumbersome and time-consuming. While there is a growing trend towards embracing open science in software engineering, it remains far from the default and, in our experience, often receives insufficient recognition in academic evaluations.

The strong focus on simulation as a research method also constitutes a limitation, as simulations inherently rely on abstractions and assumptions that may oversimplify the fine-grained and deeply human nature of communication, as well as the complex social and organizational dynamics of collaborative software engineering in general. As a result, certain nuances, such as minor interactions (for example, exchanges of emoticons in pull requests that we purposefully excluded from analysis), implicit social norms, or the interdependence between code review and the guidelines that govern it—including how strictly those guidelines are followed—may not be fully captured within the simulated environment, even though such factors shape code review as a human-centered activity. Future work is needed to broaden the configuration of both observational and simulation-based studies to capture a wider range of behaviors, contexts, and interactions that influence how information diffuses in practice.

In retrospect, one of the most significant limitations of this thesis is its limited engagement with epistemology. What constitutes knowledge in software engineering? What types of evidence justify a theoretical claim? To what extent can knowledge derived from simulations or observational studies be generalized? And what does it mean for a theory in software engineering to be considered true, useful, or explanatory? These foundational questions remain underexamined in this work, yet they are central to understanding the nature, justification, and scope of theoretical contributions. Too

late did we[2] understand the relevance and importance of these epistemological dimensions of this thesis. Although such philosophical questions may initially seem outside the scope of software engineering research, they are in fact essential, particularly because software engineering is still a young and evolving discipline. Addressing these questions more explicitly from the outset would have enriched the theoretical reflection in this thesis and strengthened its contribution to the broader discourse on theory building in software engineering.

Although this thesis maintains a strong focus on theory validation, we demonstrated the practical relevance of the theory in the context of tax compliance. With our work, we uncovered the potential risk of cross-border collaboration in the context of tax compliance for multinational enterprises. Litigation such as the case between Microsoft and the US tax authorities (IRS) in which the IRS alleges that Microsoft owes an additional $28.9 billion in tax from 2004 to 2013 [124], highlights two key implications:

1. The risk associated with cross-border collaboration, as identified in our research, is no longer abstract; it has become concrete and carries a multi-billion dollar price tag.

2. In contexts where billions of dollars and legal accountability are at stake, rigorous theory building and theory evaluation are essential. Vague assumptions, conventional wisdom, or *ad hoc* reasoning are no longer sufficient.

The story of code review has not yet been told; this thesis begins that narrative. We highlight this with three ideas for future work.

First, as revealed in our survey (cf. Chapter 7), many practitioners anticipate that AI will play a major role in code review in the future. This raises critical questions about the erosion of understanding, accountability, and trust in the code review. Future research can explore how generative AI, such as large language models (LLMs), may impact the dynamics of code review and how these changes might affect the collaborative and responsible nature of software engineering. We have only started this discussion.

Second, code review might not only be a communication network, but may also function as a decentralized, distributed, self-organizing information repository, continuously maintained through developer interaction. Instead of relying on a centralized system, knowledge about the software is embedded within the developer network and is incrementally updated through ongoing communication. A promising direction for future research can be the exploration of code review as a decentralized, distributed, self-organizing information repository where developers locate relevant information without engaging in direct communication, but retrieving archived information persis-

[2]The pronoun “we” is used throughout for consistency; however, it primarily refers to the author, whose intellectual capacity for a substantive engagement with the philosophy of science was regrettably limited.

tently stored in code review. In the same way, code review may act as a gatekeeping mechanism, offering new perspectives on the structure and function of communication networks in software engineering.

Third, we propose to apply the theory in the context of software engineering education. If code review functions as a powerful and scalable communication network in professional settings as we have shown in this research, it may serve a similar role in educational environments. Peer-based code review among students could (a) support their education in both the technical and collaborative aspects of software engineering and (b) provide a scalable communication network through which they can exchange insights in ways that are accessible and meaningful to their peers. This presents a promising research avenue to understand through our understanding of code review as a communication network how peer interaction in code review supports learning, fosters shared understanding, and helps students develop practical judgment in software development [64].

1.6 Conclusion

So, is code review a cargo cult? No, it is not. Our research has shown that code review is a powerful, scalable communication network that is capable of spreading information both widely and quickly among its participants and facilitates information diffusion across social, organizational, and architectural boundaries. By formalizing and empirically evaluating the theory of code review, this dissertation contributes to the theoretical foundation of code review as core practice in collaborative software engineering practices. Our research enhances our understanding of code review not purely as gatekeeping and quality assurance mechanisms but also as a communication network.

We also demonstrated the practical relevance of this theory by applying it in the context of tax compliance, uncovering potential financial risks for multinational enterprises due to cross-border collaboration and opening a new avenue for interdisciplinary research. We also explored the future of code review as a communication network and its role in future collaborative software engineering.

Apart from our contributions to a deeper understanding of code review and its role for collaborative software engineering, we also demonstrated that simulations as an empirical method can enhance software engineering research by providing a controlled, reproducible research method to experiment with complex systems—like the code review systems with thousands of developers—that are inherently difficult to control and, therefore, experiment with. While still uncommon, these *in silico* experiments, through explicit modelling and full reproducibility, hold significant untapped potential to advance software engineering.

1

One hope we associate with the full disclosure of the software and data used in this dissertation is to empower the research community to replicate, scrutinize, and extend our work. By making our research transparent and our artifacts openly available, we aim to contribute to a more cumulative and collaborative model of scientific progress. We therefore hope that this thesis not only deepens the understanding of code review but also encourages a broader commitment to theory-oriented research that bridges the gap between abstraction and application and helps shape a more reflective, evidence-based future for software engineering.

2

Modelling Information Diffusion in Code Review

This chapter is based on Michael Dorner, Darja Šmite, Daniel Mendez, Krzysztof Wnuk, and Jacek Czerwonka. "*Only Time Will Tell: Modelling Information Diffusion in Code Review with Time-Varying Hypergraphs*". In: *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, September 2022, pp. 195–204. DOI: 10.1145/3544902.3546254.

Abstract

Background: Modern code review is expected to facilitate knowledge sharing: All relevant information, the collective expertise, and meta-information around the code change and its context become evident, transparent, and explicit in the corresponding code review discussion. The discussion participants can leverage this information in the following code reviews; the information diffuses through the communication network that emerges from code review. Traditional time-aggregated graphs fall short in rendering information diffusion as those models ignore the temporal order of the information exchange: Information can only be passed on if it is available in the first place.

Aim: This manuscript presents a novel model based on time-varying hypergraphs for rendering information diffusion that overcomes the inherent limitations of traditional, time-aggregated graph-based models.

Method: In an in-silico experiment, we simulate an information diffusion within the internal code review at Microsoft and show the empirical impact of time on a key characteristic of information diffusion: the number of reachable participants.

Results: time-aggregation significantly overestimates the paths of information diffusion available in communication networks and, thus, is neither precise nor accurate for modelling and measuring the spread of information within communication networks that emerge from code review.

Conclusion: Our model overcomes the inherent limitations of traditional, static or time-aggregated, graph-based communication models and sheds the first light on information diffusion through code review. We believe that our model can serve as a foundation for understanding, measuring, managing, and improving knowledge sharing in code review in particular and information diffusion in software engineering in general.

2.1 Introduction

Code review has transformed over the last decades from a waterfall-like procedure primarily used for detecting bugs in formal, heavyweight code inspections in the 1980s to a knowledge-sharing platform in an informal, tool-supported, lightweight process nowadays [7, 14, 17]. Since modern software systems are often too large, too complex, and evolving too fast for a single developer to oversee all parts of the software and, therefore, to understand all implications of a change, most software projects use code review to foster a broad discussion on the change and its impact before it is merged into the codebase. Each code review becomes a communication channel to share knowledge among the discussion participants: All relevant information, collective expertise, and meta-information about the change become evident, transparent, and

explicit through those discussions and are shared among the participants. Since the participants implicitly cache this information, they can use, build upon, and spread it in the upcoming code reviews they participate in. Over time, information is spread through code review among its participants, the so-called *information diffusion*.

Until today, software engineering research relies on time-aggregated graph-based models for representing communication networks of all kinds [60]. However, time-aggregated, graph-based communication models are not capable of rendering such an information diffusion since information diffusion is neither necessarily bilateral nor instant: In a discussion during a code review, multiple people can receive information concurrently and information can only be passed on if it is available to the participant beforehand. Depending on the temporal availability of vertices and edges, the information takes different routes through the communication network—a type of network topology traditional, time-aggregated graph-based communication models cannot render.

Motivated by these shortcomings, we introduce a novel model for information diffusion in channel-based communication based on time-varying hypergraphs to research how information originated from a code review discussion spread among the communication participants. We validate our time-respecting model in comparison with an equivalent but a time-aggregated graph-based model in a computer simulation that empirically shows the impact and importance of time-awareness for information diffusion analysis in code review. For this comparison, we use a key characteristic for information diffusion, the number of reachable participants, which also reflects the number of paths in a communication network that are valid and available for direct and indirect information exchange.

The main contributions of this manuscript are as follows:

- We introduce a novel communication model based on time-varying hypergraphs for information diffusion within communication networks.
- To this end, we provide a concise and gentle introduction to the mathematical foundation of time-dependent hypergraphs and the impact of topological and temporal distance on information diffusion modelling.
- We simulate the spread of information within the communication network emerging from code review at Microsoft to validate our model compared to an equivalent but time-aggregated model concerning the number of reachable participants for each participant.
- We present first insights on the theoretical maximum spread of information possible within the communication network emerging from code review: the number of reachable participants.

- We highlight possible probabilistic extensions to and future applications of our model as a proxy for the capacity of code review as a knowledge-sharing platform.

The manuscript is structured as follows: We begin with a gentle mathematical introduction to time-varying hypergraphs in Section 2.2. In Section 2.3, we provide an overview of state of the art on graph-based communication models in software engineering and related disciplines, as well as *in silico* experiments and simulation in software engineering. We formalize code review as channel-based communication to the extent we deem necessary and define our conceptual and computer model in Section 2.4. In Section 2.5, we showcase and validate our model in a computer simulation rendering an artificial information diffusion in an empirical communication network that emerges from code review at Microsoft. After we present the resulting comparison of the time-ignoring and time-respecting reachable participants in an information diffusion simulation in Section 2.6 and discuss the findings in Section 2.7, the manuscript closes with a conclusion and future work in Section 2.8.

2.2 A Gentle Introduction to Time-Varying Hypergraphs

In this work, we combine two lesser-known graph-theoretical concepts: time-variance of graphs and hypergraphs. We follow the definitions and notation by [24] for time-varying graphs and by [90] for hypergraphs to a large extent.

A *time-varying graph* is a graph whose edges (and vertices) are active or available only at specific points in time. A *hypergraph* is a generalization of a graph in which an edge (a so-called *hyperedge*) can connect any arbitrary number[1] of vertices.

Thus, a time-varying hypergraph is a hypergraph which hyperedges (and vertices) are time-dependent. Mathematically, a time-varying hypergraph is a quintuple $\mathcal{H} = (V, \mathcal{E}, \rho, \xi, \psi)$ where

- V is a set of vertices,
- $\mathcal{E}$ is a set of hyperedges connecting any number of vertices,
- ρ is the *hyperedge presence function* indicating whether a hyperedge is active at a given time,
- $\xi \colon E \times \mathcal{T} \to \mathbb{T}$ is the *latency function* indicating the duration to cross a given hyperedge,
- $\psi \colon V \times \mathcal{T} \to \{0, 1\}$ is the *vertex presence function* indicating whether a given vertex is available at a given time, and

[1] A classical graph is a subclass of a hypergraph with hyperedges that always connect exactly two (in case of self-loops not necessarily distinct) vertices.

- $\mathcal{T} \in \mathbb{T}$ is the lifetime of the system.

The temporal domain $\mathbb{T}$ is generally assumed to be $\mathbb{N}$ for discrete-time systems or $\mathbb{R}_+$ for continuous-time systems.

Because the edges are time-dependent, the walk through a (hyper)graph is also time-dependent. Formally, a sequence of tuples

$$\mathcal{J} = (e_1, t_1), (e_2, t_2), \ldots, (e_k, t_k),$$

such that $e_1, e_2, \ldots, e_k$ is a walk in $\mathcal{H}$, is a *journey* in $\mathcal{H}$ iff $\rho(e_i, t_i) = 1$ and $t_{i+1} > t_i + \xi(e_i, t_i)$ for all $i < k$.[2] Additional constraints maybe required in specific domains of application, such as the condition $\rho_{[t_i, t_i+\xi(e_i,t_i)]}(e_i) = 1$: the hyperedge remains present until the hyperedge is crossed.

We define $\mathcal{J}^*_{\mathcal{H}}$ the set of all possible journeys in a time-varying graph $\mathcal{H}$ and $\mathcal{J}^*_{(u,v)} \in \mathcal{J}^*_{\mathcal{H}}$ the journeys between vertices u and v. If $\mathcal{J}^*_{(u,v)} \neq \emptyset$, u can reach v, or in short notation $u \rightsquigarrow v$. In general, journeys are not symmetric and transitive—regardless of whether the hypergraph is directed or undirected: $u \rightsquigarrow v \not\Leftrightarrow v \rightsquigarrow u$. Given a vertex u, the set $\{v \in V : u \rightsquigarrow v\}$ is called *horizon* of vertex u.[3]

A time-varying hypergraph $\mathcal{H}$ can be transformed in an equivalent bipartite graph $B = (V, \mathcal{E}, E, \psi)$ where

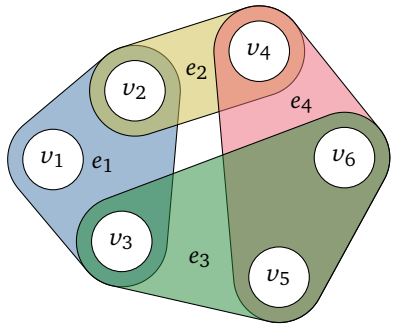
- V is the set of vertices from the equivalent hypergraph,
- $\mathcal{E}$ is the set of hyperedges from the equivalent hypergraph,
- V and $\mathcal{E}$ are disjunct ($V \cap \mathcal{E} = \emptyset$) and both vertices of the bipartite graph,
- $E = \{(v, e) \mid u \in V, e \in \mathcal{E}\}$ are the vertices of the bipartite graph that connect vertices V with hyperedges $\mathcal{E}$, and
- ψ is the edge presence function for the vertices $\mathcal{E}$ reflecting the edge presence function ρ of the time-varying hypergraph such that $\psi(e) = \rho(e), e \in \mathcal{E}$.

Although an equivalent bipartite graph can represent a hypergraph, both concepts are semantically different. For an in-depth mathematical discussion, we refer the reader to the work by [90].

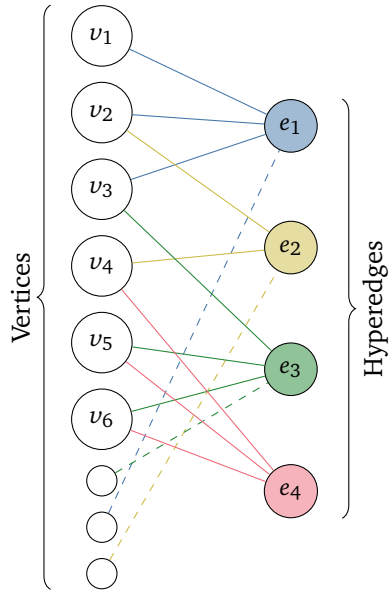
Figure 2.1 provides an example of a time-varying hypergraph and its transformation to an equivalent bipartite graph: The colors of the hyperedges reflect the colors of the righthand vertices in the bipartite graph. Furthermore, the example also shows

[2] We deviate from Casteigts et al. [24] who require $t_{i+1} \geq t_i + \xi(e_i, t_i)$.

[3] The horizon of a vertex v in a time-varying graph is not equivalent to the connected component that contains the vertex v (neither strongly nor weakly component of the time-varying graph [84]): the horizon is neither a reflexive (i.e. horizon of vertex v does not necessarily contain the vertex v.) nor a symmetric relation.



(a) An example time-varying hypergraph, a generalization of a graph which edges, the so-called hyperedges, (denoted by $e_\square$) can link any arbitrary number of vertices (denoted by $v_\square$): For example, hyperedge e_3 connects four vertices. The reachability (or information diffusion in our case) of vertex v_1 depends highly on the temporal order of the hyperedges: if $e_1 < e_2 < e_4 < e_3$, the resulting horizon contains all vertices; if $e_1 > e_2 \geq e_3$ no information can be spread because no time-respecting path (journey) is available.



(b) Any hypergraph can be transformed into an equivalent bipartite graph: The hyperedges and the vertices from the time-varying hypergraph from Figure 2.1a become the two distinct sets of vertices of a bipartite graph.

Figure 2.1: A simple example of a hypergraph and its bipartite-graph equivalent.

the impact of time on the horizon of vertices in such hypergraphs. Depending on the presence of the hyperedges, there are different journeys from the vertices v_1 to v_6. Please mind that there is no time-respecting path (journey) in the opposite direction from v_6 to v_1.

2.3 Background

This section discusses related work on modelling communication in software engineering and provides context for our research approach, *in silico* experiments and simulations.

2.3.1 Modelling Communication in Software Engineering Research

To the best of our knowledge, our modelling approach for communication using time-dependent hypergraphs has not previously been used in software engineering research

and other disciplines. time-dependent hypergraphs are first applied in the research context of epidemiology: Independent of us and in parallel to our work, Antelmi et al. first defined and used time-depending hypergraphs to show the importance of time on disease diffusion. Neither the domain of epidemiology—information does not spread like viruses—nor the representation of hyperedges map to our research: In their model, hyperedges refer to geo-locations that are constant over time and vertices to persons meeting at those geo-locations over time. Hyperedges in our model reflect the channels of the information exchange and are time-dependent.

Software engineering research uses traditional graphs to model different types of information exchange and the networks that emerge from that communication. Herbold et al. identified in their systematic mapping study 182 studies researching various networks of developers modelled as graphs. We found that all studies use time-aggregated graphs to model developer interactions. Those limitations make time-aggregated graphs incapable of rendering time-dependent phenomena without introducing a large error by this simplification.

However, the use of time-respecting network models and the research on information diffusion in software engineering is new but not wholly unexplored.

Lamba et al. used a multi-layered time-dependent graph for investigating the tool diffusion of 12 quality assurance tools within the *npm* ecosystem—without explicitly using this terminology. Although there are several similarities to our work at first sight, the used theoretical framework on the diffusion of innovations by Rogers does not apply in the general case of communication as Rogers states: Diffusion of innovations is "a special type of communication, in that the messages are concerned with new ideas. Communication is a process in which participants create and share information to reach a mutual understanding." [104] Since we are modelling the exchange of information in general without any prior knowledge of its novelty value, the theory framework by Rogers does not apply to our research. Therefore, we explicitly use the term information diffusion in this study.

Nia et al. investigated edge transitivity and the introduced error through the aggregation over time. They showed for the mailings lists of three open-source projects (Apache, Perl, and MySQL) that the clustering coefficient and the 2-path counts are robust to data aggregation across large intervals (over one year) even though such aggregation may lead to transitive faults. However, the results are only valid for time-aggregated systems. This implies that the findings do not apply to our research on and the modelling of information diffusion as the spread of information is a highly dynamic, time-dependent process.

Gote, Scholtes, and Schweitzer analyzed the temporal co-editing networks in software development teams using a rolling window approach [55]. In detail, the study uses time-stamped bipartite graphs to model the relationship between developers and edited files. Since hypergraphs can be represented by bipartite graphs where

hyperedges and vertices are the two distinct sets of vertices, the modelling approach is quite similar. However, the team converted this bipartite graph into a directed, acyclic graph (DAG) representing a sequence of consecutive co-editing relations of developers editing the given file to estimate knowledge flow. The nodes in this DAG represent commits and edges co-editing relationship between the authors of the commits. The connected components[4] of the DAG represent proxies of knowledge flow, what we call information diffusion. Although this modelling approach respects the temporal order, the DAG cannot reflect the temporal distance and, thus, does not allow insights into how much time has left during the diffusion process. Only the topological distance (how many hops between two vertices) is available. The temporal distance (how much time has passed), however, reveals key characteristics of information since information ages constantly and information not delivered at the right time is outdated or simply invalid. The results always refer to the observation window but no more fine-grained insights. This shortcoming applies to all modelling approaches using any type of directed graph representing the order.

A similar problem occurs with models based multigraphs for representing the parallel connection of multiple nodes. Although technically possible, multigraphs blur the relationship between an edge and a communication channel (i.e., code review): a communication channel would no longer correspond to one edge but a set of edges.

2.3.2 In Silico Experiments and Simulations in Software Engineering Research

In our study, we conduct an *in silico* experiment. In contrast to in-vivo, in-vitro, and in-virtuo experiments, an *in silico* experiment is performed solely via a computer simulation. Both subjects and the real world are described as simulation models [63]. Any human interaction is reduced to a minimum. Those simulation models are like virtual laboratories where hypotheses about observed problems can be tested, and corrective policies can be experimented with before they are implemented in the real system [81].

The use of the term *simulation* varies substantially, from discipline and context [48]. In this work, we rely on the definition by Banks et al.: A simulation is the imitation of the operation of a real-world process or system over time. The behavior of a system as it evolves over time is studied by developing a simulation model, a purposeful abstraction of a real-world system in the form of a set of assumptions concerning the system's operation.

Although simulation models have been applied in different research fields of software engineering, e.g., process engineering, risk management, and quality assurance [81], due to the need for a large amount of knowledge, *in silico* studies are scarce in software engineering [63].

[4]We assume the model uses the *strongly connected components* since the *connected components* only exist in undirected graphs.

2.4 Modelling Information Diffusion with Time-Varying Hypergraphs

Communication is a complex and manifold process that changes over time. We need models as a purposeful and simplified abstraction of such complex phenomena, imitating those complex real-world processes to enable measurability, gain insights, predict outcomes, and understand the mechanics.

A simulation model has two components: a conceptual model and a computer model [49]. A conceptual model is a (non-software) abstraction of the simulation model that is to be developed, describing objectives, inputs, outputs, content, assumptions, and simplifications of the model [103]. On the other hand, the computer model describes the conceptual model implemented in software.

In the following subsection, we define and discuss the conceptual and computer model of information diffusion in code review.

2.4.1 Conceptual Model

Communication, the purposeful, intentional, and active exchange of information among humans, does not happen in the void. It requires a channel to exchange information. A *communication channel* is a conduit for exchanging information among communication participants. Those channels are

1. *multiplexing*—A channel connects all communication participants sending and receiving information.
2. *reciprocal*—The sender of information also receives information and the receiver also sends information. The information exchange converges. This can be in the form of feedback, queries, or acknowledgments. Pure broadcasting without any form of feedback does not satisfy our definition of communication.
3. *concurrent*—Although a human can only feed into and consume from one channel at a time, multiple concurrent channels are usually in use.
4. *time-dependent*—Channels are not always available; the channels are closed after the information is transmitted.

Channels group and structure the information for the communication participants over time and content. Over time, the set of all communication channels forms a communication network among the communication participants.

In the context of researching the information diffusion in this study, a communication channel is a discussion in a merge (or pull) request. A channel for a code review on a merge request begins with the initial submission and ends with the merge in case of an acceptance or a rejection. All participants of the review of the merge request feed information into the channel and, thereby, are connected through this channel and exchange information they communicate. After the code review is completed

and the discussion has converged, the channel is closed and archived, and no new information becomes explicit and could emerge. However, a closed channel is usually not deleted but archived and is still available for passive information gathering. We do not intend to model this passive absorption of information from archived channels by retrospection with our model.

From the previous postulates on channel-based communication in software engineering, we derive our computer model: Each communication medium forms an undirected, time-varying hypergraph in which hyperedges represent communication channels. Those hyperedges are available over time and make the hypergraph time-dependent. Additionally, we allow parallel hyperedges[5]—although unlikely, multiple parallel communication channels can emerge between the same participants at the same time but in different contexts.

Such an undirected, time-varying hypergraph reflects all four basic attributes of channel-based communication:

- *multiplexing*—since a single hyperedge connects multiple vertices,
- *concurrent*—since (multi-)hypergraphs allow parallel hyperedges,
- *reciprocal*—since the hypergraph is undirected, information is exchanged in both directions, and
- *time-dependent*—since the hypergraph is time-varying.

In detail, we define our model for information diffusion in an observation window $\mathcal{T}$ to be an undirected time-varying hypergraph

$$\mathcal{H} = (V, \mathcal{E}, \rho, \xi, \psi)$$

where

- V is the set of all human participants in the communication as vertices
- $\mathcal{E}$ is a multiset (parallel edges are permitted) of all communication channels as hyperedges,
- ρ is the *hyperedge presence function* indicating whether a communication channel is active at a given time,
- $\xi\colon E \times \mathcal{T} \to \mathbb{T}$, called *latency function*, indicating the duration to exchange an information among communication participants within a communication channel (hyperedge),

[5]This makes the hypergraph formally a *multi-hypergraph* [90]. However, we consider the difference between a hypergraph and a multi-hypergraph as marginal since it is grounded in set theory: Sets do not allow multiple instances of the elements. Therefore, instead of a set of hyperedges, we use a multiset of hyperedges that allows multiple instances of the hyperedge.

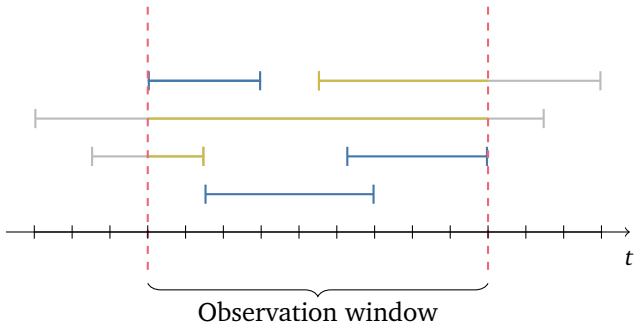


Figure 2.2: Not all communication channels started or ended within the observed time window (indicated by blue): Cut channels (indicated by yellow) are incomplete and lead to a blur at the borders of our measurements.

- $\psi : V \times \mathcal{T} \rightarrow \{0, 1\}$, called *vertex presence function*, indicating whether a given vertex is available at a given time.

Communication and the spread of information are usually ongoing, continuous processes. As for any continuous, real-world process, we only can make assumptions about windowed observations of that phenomenon. The lifetime of our system is limited by this observation window which borders induce blur in our investigations: The communication may have started before or ended after our observed time window; information is lost. Thus, we must define our model's lifetime as the observation window. Figure 2.2 illustrates this problem of the observation window for an ongoing, continuous system.

2.4.2 Computer Model

We implement the hypergraph as an equivalent bipartite graph using the widely used Python graph package *networkx* [56]: The hypergraph vertices and hyperedges become two sets of vertices of the bipartite graph. The vertices of those disjoint sets are connected if a hypergraph edge was part of the hyperedge. For a more detailed and graphical description of the equivalence of hypergraphs and bipartite graphs, we refer the reader to Section 2.2.

To ensure that the computational model accurately represents the underlying mathematical model and its solution, we applied four quality assurance approaches in the model verification phase:

- **Code walk-throughs**—We independently conducted code walk-throughs through the simulation code with three Python and graph experts.
- **High test-coverage**—The simulation code has a test coverage of about 99%.

- **Code readability and documentation**—We provide comprehensive documentation on the usage and design decisions to enable broad use and further development. We followed the standard Python programming style guidelines PEP8 for readability.
- **Publicly available and open source**—The model parameterization and simulation code [31] as well as all intermediate and final results [32] are publicly available.

2.5 Experimental Design

In this section, we describe the simulation as an *in silico* experiment [46] that evaluates the impact of ignoring and respecting time in a temporal graph for modelling information diffusion in code review. The purpose of this simulation is two-fold: We provide a proof of concept of our modelling approach and present a first validation by comparison to another model [48]. Through this comparison, the impact of time on communication networks becomes evident.

In this computer simulation, we measure the number of individuals receiving information from a code review directly and indirectly in a best-case scenario.

Figure 2.3, we present a high-level overview of our simulation.

In the following subsection, we describe our simulation assumptions (Section 2.5.1), the parametrization of our model using empirical data from code review at Microsoft (Section 2.5.2), and the simulation mathematically and algorithmically (Section 2.5.3).

2.5.1 Assumptions

For this study, we made the following assumptions for information diffusion in code review:

- *Channel-based*—Information can only be exchanged along the information channels.
- *Perfect caching*—All code review participants can remember and cache all information in all code reviews they participate in within the considered time frame.
- *Perfect diffusion*—All participants instantly pass on information at any occasion in all available communication channels in code review.
- *Information diffusion only in code review*—For this simulation, we assume that information gained from discussions in code review diffuses only through code review.

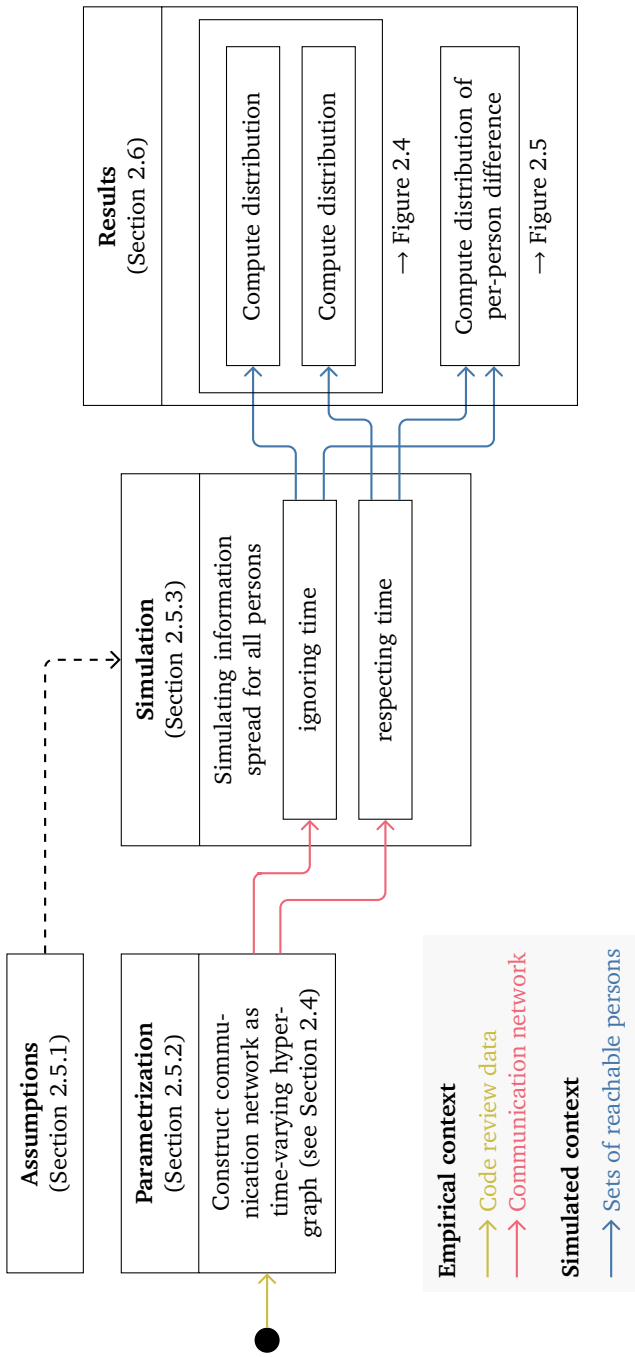


Figure 2.3: An overview of the simulation.

- *Information availability*—To have a common starting point and make the results comparable, the information to be diffused in the network is already available to the participant that is the origin of the information diffusion process.

We discuss the impact of those assumptions in Section 2.7. The assumption $\psi \to 1$, meaning all code review participants are available over the considered time-frame $\mathcal{T}$ of four weeks, is implicit and does not impact the measurements: If a participant is either inactive or removed temporarily or permanently from the communication network has no impact on the number of reachable participants.

Our assumptions make the number of reachable participants a best-case scenario and do not likely represent an empirical information diffusion process. However, the relative comparison is adequate since all assumptions are equal for time-ignoring and time-respecting information diffusion measurements.

2.5.2 Parametrization

To parametrize the model, we extracted all internal, human code review interactions tracked by Microsoft's internal code review tool *CodeFlow* [18] run by Azure DevOps service. Although not Microsoft's only code review tool, it represents a large portion of the company's code review activity. All non-human code-review participants and interactions are excluded. The dataset contains all human code review interactions from 2020-02-03 to 2020-03-01, inclusively, corresponding to full four calendar weeks without significant discontinuities by public holidays such as Christmas. The time frame is arbitrary, however.

The underlying hypergraph has 37,103 vertices (developers) and 309,740 hyperedges (communication channels) for both models. We made all code and data publicly available in our replication package.

2.5.3 Simulation

In our simulation, we use our parametrized communication model to measure how many participants can be reached from each participant using either time-respecting or time-ignoring paths in the communication network.

Mathematically, the number of reachable participants is a set of vertices that can be reached from u:

$$|\{v \in V : u \rightsquigarrow v\}|$$

If reachability is time-respecting, the measure is called *horizon*. If time and the temporal order are ignored for the reachability, the horizon becomes the *connected component* containing vertex u.

Algorithmically, both measurements on the number of reachable participants for all vertices are variations of the breadth-first search. Algorithm 1 describes the time-

ignoring and time-respecting breadth-first search approach in pseudocode; our Python implementation can be found in the replication package.

Algorithm 1: Breadth-first search for vertex s of a time-varying hypergraph $\mathcal{H}$.

Input : time-varying Hypergraph $\mathcal{H} = (V, \mathcal{E}, \rho, \xi, \psi)$
Start node $s \in V$
Output : An set for all node $v \in V$ reachable of s

$Q \longleftarrow$ initialize empty queue
push $s \longrightarrow Q$
mark s as reachable

while $Q \neq \emptyset$ **do**
  pop $Q \longrightarrow v$
  $N \longleftarrow \begin{cases} N(v) & \text{if time-ignoring} \\ \{n \in N(v) \subseteq V \mid v \rightsquigarrow n\} & \text{if time-respecting} \end{cases}$
  foreach $n \in N$ ▸ *All available neighbors of s* **do**
    if *n not marked as reachable* **then**
      push $n \longrightarrow Q$
      marks n as reachable
    end
  end
end
return all reachable nodes

The algorithm is integrated into our computer model and implemented in Python. All code is publicly available [31][6] under MIT license. To ensure the correctness of the implementation, we created an extensive test setup.

2.6 Results

All statistical locations of the reachable participants (namely median, mean, min, and max) are significantly smaller in the time-respecting information diffusion than the time-ignoring information diffusion: The mean time-ignoring reachable participants are 29,660 persons, the median is 33,172 persons. This unequal distribution is caused by the symmetry characteristic of the reachability in (undirected) graphs: All vertices have the same connected component. The largest component has 33,173 persons, which is 89.41% of all persons due to the symmetry characteristic of the

[6] For more information, see also `https://github.com/michaeldorner/only-time-will-tell`.

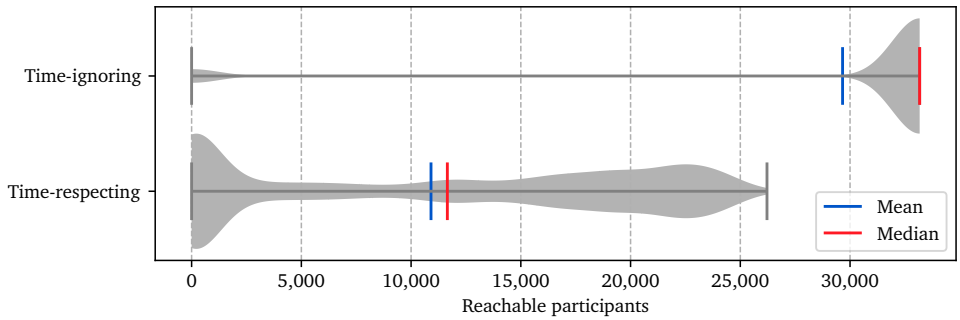


Figure 2.4: Distribution of the time-respecting and time-ignoring reachable participants for a simulated information diffusion: All statistical locations (mean, median, min, and max) are significantly smaller when respecting time.

connected component in (undirected) graphs. All other connected components are significantly smaller: The second-largest component has 108 persons (0.29%). The number of time-respecting reachable participants draws a more fine-grained picture: On average, 10,907 persons (mean) and 11,652 persons (median), respectively, can be reached. At most, 26,216 persons (70.66%) can be reached. Figure 2.4 contrasts both distributions.

The time-respecting and the time-ignoring per-person difference in the reachable participants differ significantly. In average, the difference is 18,752 persons (mean) or 16,822 persons (median). The largest per-person difference is 33,171, which is also the maximum value: while all persons are reachable when time is ignored, no person is reachable when time is considered, i.e., no path in temporal order (journey) is available. Figure 2.5 depicts the per-person difference between the time-ignoring and time-respecting reachable participants.

Both perspectives on reachable participants confirm the remarkable difference in respecting or ignoring time for measurements of information diffusion: time-ignorance overestimates the available and temporal valid paths (journeys) in communication networks. The temporal order has a significant impact on the horizon and, thus, on the paths valid for information to diffuse.

2.7 Discussion

At this point, we would like to emphasize again that the measurements do not describe an empirical information diffusion: Although the network structure is constructed by real-world data and, therefore, empirical, the resulting information diffusion, the spread of information, is simulated. However, although the spread of information is artificial and the information has never empirically reached the participants, we be-

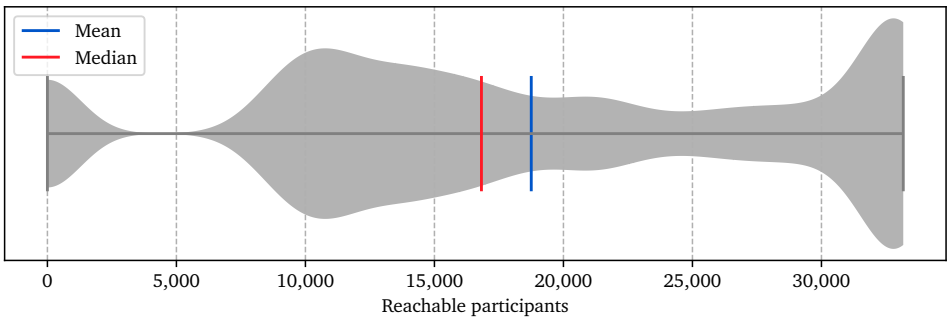


Figure 2.5: Distribution of the per-person difference of the time-ignoring and time-respecting reachable participants: All persons have a significantly larger number of reachable persons if respecting time.

lieve the maximum number of reachable participants can be considered empirical, not neglecting the constraints we put on our simulation in the forms of our assumptions.

All five assumptions described in Section 2.5 are applied to both models. The constraints apply to both measurements to the same extent. Therefore, comparing the results ignoring and respecting time is sound and adequate.

Our assumptions are not easily transferrable to other empirical investigations on information diffusion in code review or general. Both simulation assumptions of perfect caching and perfect diffusion are best-case assumptions leading to an upper bound. We strongly believe that this upper bound of reachable participants is not achievable and less meaningful in reality, particularly over larger time frames: information may get outdated, irrelevant, or even false over time. Also, human retentiveness, attention, and memory are limited. Future research can investigate the average number of reachable participants within code review and the impact of the topological and temporal distance on the probability of information diffusion.

Furthermore, information is not only diffused through code review but also through other communication media like instant messaging or virtual or in-person meetings. For a more holistic view of information diffusion in software engineering projects—not only through and within code review—, we need to capture more communication networks (e.g., code review, instant messaging, e-mail, classical meetings) stacked on each other to capture all possible information diffusion journeys. Figure 2.6 gives an example of stacked communication networks consisting of overlapping hypergraphs.

The second advantage of our model to be capable of rendering interconnections between more than two persons is not further discussed yet. 33.98% of all code reviews at Microsoft involves more than two persons and, thus, cannot be captured by classical graphs, only by multigraphs having parallel edges. However, models with multigraphs—although technically possible—blur the relationship between an edge

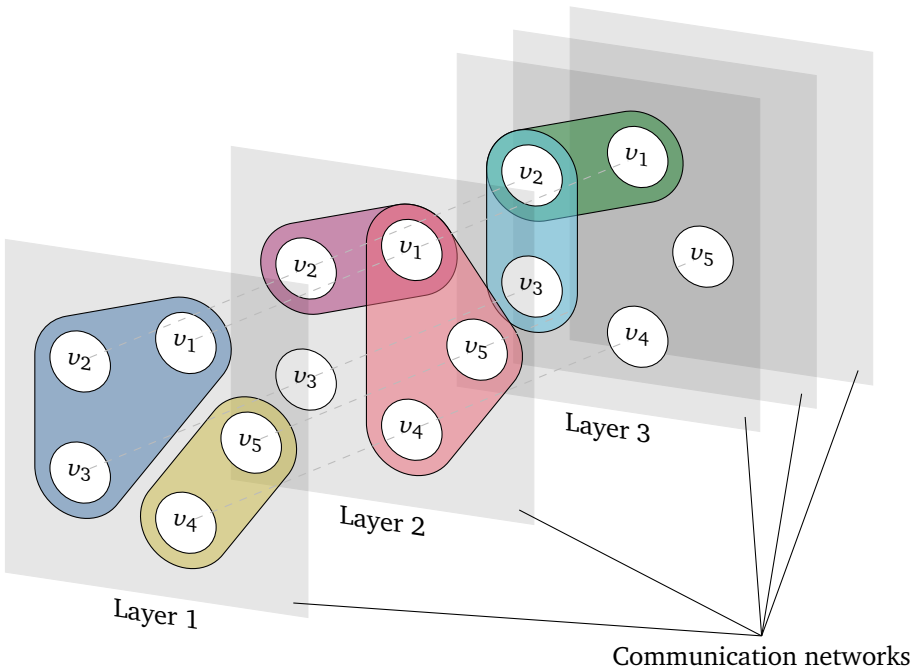


Figure 2.6: For a more holistic view of information diffusion, different communication network layers are required.

and a communication channel (i.e., code review): a code review would no longer correspond to one (hyper)edge but a set of edges. The graph becomes less expressive and more complex to compute and simulate.

2.8 Conclusion

We present a model based on time-varying hypergraphs for modelling and analyzing information diffusion within code review. The model overcomes the limitations of existing graph-based models and enables research on time-respecting and multilateral information diffusion.

Our simulation based on the code review at Microsoft to estimate the empirical impact of time-dependency on information diffusion reveals that significantly fewer code review participants are reachable and, therefore, significantly fewer paths to diffuse information are valid if time is respected. Ignoring time in communication networks introduces a large error since the time-ignoring model overestimates the available and valid paths within such communication networks.

We believe that the available information diffusion paths, as well as the topological (measured in the number of time-respecting hops) or temporal distance (measured in time) between participants revealed by our model, provide a solid foundation for future research on the capacity of code review as a knowledge-sharing platform as suggested by prior qualitative studies [7, 14, 17].

Our model can be easily extended by probability, an integral part of information diffusion: not every information is spread on every occasion: *random or probabilistic time-varying graphs* with an edge presence function $\rho\colon E \times \mathcal{T} \to [0,1]$ or vertex presence function $\psi\colon V \times \mathcal{T} \to [0,1]$ allows to render probabilistic processes of information diffusion and estimate the stability of communication networks. As a generalization of a traditional graph, hypergraphs are a promising modelling tool for not only communication networks but also other higher-order systems since they are compatible with traditional graph metrics and algorithms.

To enable researchers and practitioners to replicate, reproduce, and extend our work and model, we provide an extensive replication package containing all code [31] and data [32][7]. We also explicitly encourage researchers from outside software engineering to apply, revise, and advance our model.

Acknowledgements

We thank Dietmar Pfahl for his valuable contributions to the presentation of the paper and Andreas Bauer and Ehsan Zabardast for their feedback which improved the simu-

[7] For more information, see also `https://github.com/michaeldorner/only-time-will-tell`.

lation code greatly. We also thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and helpful suggestions.

This work was supported by the KKS Foundation through the SERT project (research profile grant 2018/010) at Blekinge Institute of Technology.

3 Estimating Information Diffusion in Closed-Source Code Review

This chapter is based on Michael Dorner, Daniel Mendez, Krzysztof Wnuk, Ehsan Zabardast, and Jacek Czerwonka. "*The Upper Bound of Information Diffusion in Code Review*". In: *Empirical Software Engineering* (June 2023).

Abstract

Background: Code review, the discussion around a code change among humans, forms a communication network that enables its participants to exchange and spread information. Although reported by qualitative studies, our understanding of the capability of code review as a communication network is still limited.

Objective: In this article, we report on a first step towards understanding and evaluating the capability of code review as a communication network by quantifying how fast and how far information can spread through code review: the upper bound of information diffusion in code review.

Method: In an *in silico* experiment, we simulate an artificial information diffusion within large (Microsoft), mid-sized (Spotify), and small code review systems (Trivago) modelled as communication networks. We then measure the minimal topological and temporal distances between the participants to quantify how far and how fast information can spread in code review.

Results: An average code review participants in the small and mid-sized code review systems can spread information to between 72 % and 85 % of all code review participants within four weeks independently of network size and tooling; for the large code review systems, we found an absolute boundary of about 11 000 reachable participants. On average (median), information can spread between two participants in code review in less than five hops and less than five days.

Conclusion: We found evidence that the communication network emerging from code review scales well and spreads information fast and broadly, corroborating the findings of prior qualitative work. The study lays the foundation for understanding and improving code review as a communication network.

3.1 Introduction

Modern software systems are often too large, too complex, and evolve too fast for a single developer to oversee all parts of the software and, thus, to understand all implications of a change. Therefore, most software projects rely on code review to foster discussions on changes and their impacts before they are merged into the codebases to assure and maintain the quality of the software system. All available and required information about a change can become evident, transparent, and explicit through those discussions and can be shared among the participants. The discussion participants can leverage this information for their own work and pass it on in the following code reviews; the information diffuses through the communication network that emerges from code review.

Five qualitative studies have so far reported on the transition of code review from a more waterfall-like procedure used for detecting bugs in formal, heavyweight code

inspections as done in the 1980s towards a more informal, tool-supported, and lightweight communication network for developers to provide and receive relevant and context-specific information for the code change [7, 14, 17, 99, 106].

Available qualitative studies strengthen already our confidence in the motivation for and expectation towards modern code review as a communication network. However, there is still little to no research that has quantified and measured the actual capability of code review as a communication network. In this article, we report on our experiment results that complement and corroborate those five available qualitative studies.

The objective of our study is to make a first step towards better understanding and evaluating code review as a communication network by quantifying how far and how fast information can spread among the participants in code review.

In detail, we set out the following two research questions to answer:

RQ 1 How far can information spread through code review?

RQ 2 How fast can information spread through code review?

We address those two research questions in an *in silico* experiment that simulates an artificial information diffusion in code review networks at three industry cases of different sizes and different code review tools: Microsoft, Spotify, and Trivago. The simulated information diffusion within the communication networks identifies all minimal time-respecting paths reflecting information diffusing through the communication network under best-case assumptions. The participants along those minimal time-respecting paths describe how far information can spread among code review participants (RQ 1), and the minimal topological and temporal distances between participants describe how fast information spreads (RQ 2). Both measures together allow us to better understand the upper bound of information diffusion in code review.

The main contribution of our study is an *in silico* experiment to simulate information diffusion within three industrial code review systems to provide a quantitative assessment of code review as a communication network under best-case assumptions. Beyond this main contribution, we also synthesize qualitative findings from prior work regarding the expectations and motivations towards code review as motivation for our work and provide an extensive and thoroughly engineered replication package.

For this article, we define code review as the informal and asynchronous discussion around a code change among humans. This means older results from formal code inspections and pair programming as an informal but synchronous discussion around a code change among usually two developers are beyond the scope of our study.

In our study, we focus on code review in an industrial context. Although code review is nearly omnipresent in open source as well and the results are not necessarily contradicting, we strongly believe that results and findings from open source, such

as [92, 101] and [100], are not directly transferable to industrial settings without further considerations. The mechanics and incentives in open source differ, and so do the organizational structure, liability, and commitment [11].

The remainder of this paper is structured as follows: In Section 3.2, we provide an overview of the state of the art on the expectation towards code review, measuring information exchange in code review, information diffusion, and simulation as empirical research method. Section 3.3 describes our simulation model (Section 3.3.1) and its empirical parametrization (Section 3.3.2) in detail. After we report and discuss the simulation results in Sections 3.4 and 3.5, we discuss the limitations of our work in Section 3.6 and close our article with a conclusion and outlook on future work in Section 3.7.

3.2 Background

[8] identified different research themes on code review in a large systematic mapping study where the authors analyzed 244 primary studies until 2021 (inclusive). They further assessed the practitioners' perceptions on the relevance of those code review research themes through a survey of 25 practitioners. 68% of the practitioners from the survey mentioned the importance of conducting research on a more differentiated view of improvements through code review going beyond finding defects. Our research aims to fill that gap.

In the following, we elaborate on background and related work with special attention to synthesizing existing qualitative studies. We explore, in particular, the expectations towards code review in industry before laying the foundation for our simulation study by discussing measurements in information exchange, information diffusion, and, more generally, simulations as an empirical research method.

3.2.1 Expectations Towards Code Review in Industry

Although [82] report on preliminary results of a systematic mapping study on the expected benefits of code review, we could not reconstruct how and why the proposed themes—in particular those for knowledge sharing—map the referenced work. Moreover, the study does not distinguish between the expectations towards code review in an open-source and an industrial setting which, as we argued, are not necessarily comparable due to the differences in incentives, organizational structure, liability, and commitment.

In this section, we, therefore, concentrate on discussing and synthesizing five qualitative studies which have investigated the motivations and expectations towards code review in an industrial context: [7, 14, 17, 29, 106]. Table 3.1 summarizes the findings among the five prior qualitative work and their definition.

Table 3.1: Expectations towards code review reported in [7, 14, 17, 29, 106]

Identifier	Expectation	Definition
[7] → 1	Finding defects	Without explicit definition, presumably comments or changes on correctness or defects in alignment with the [7] → 2.
[7] → 2	Code improvements	"Comments or changes about code in terms of readability, commenting, consistency, dead code removal, etc., [without comments or changes] on correctness or defects."
[7] → 3	Alternative solutions	"Changes and comments on improving the submitted code by adopting an idea that leads to a better implementation."
[7] → 4	Team awareness and transparency	Without explicit definition, improved information flow across team boundaries.
[7] → 5	Share code ownership	Without explicit definition.
[7] → 6	Knowledge sharing (or learning)	Without explicit definition.
[14] → 1	Finding defects	Improved external code quality.
[14] → 2	Better code quality	Improved internal code quality.
[14] → 3	Finding better solutions	Finding new or better solutions.
[14] → 4	Sense of mutual responsibility	Improved collective code ownership and solidarity.
[14] → 5	Compliance to QA guidelines	Compliance to standards or regulatory norms.
[14] → 6	Learning (reviewer)	Learning for the author of the code change.
[14] → 7	Learning (author)	Learning for the reviewer of the code change.
[17] → 1	Maintainability	"Legibility, testability, adherence to style guidelines, adherence to application integrity, and conformance to project requirements."
[17] → 2	Knowledge sharing	"Code review facilitates multiple types of knowledge sharing." "Code review interactions help both authors and reviewers learn how to solve problems [...]."
[17] → 3	Functional defects	Eliminating "logical errors, corner cases, security issues, or general incompatibility problems."
[17] → 4	Community building	Without further definition.
[17] → 5	Minor errors, typos	Without further definition.

Identifier	Expectation	Definition
[106] → 1	Accident prevention	Avoiding the introduction of bugs, defects, or other quality-related issues.
[106] → 2	Gatekeeping	"Establishment or maintenance of boundaries around source code, design choices, or other artifacts."
[106] → 3	Maintaining norms	"Organization preference for a discretionary choice, e.g., formatting or API usage patterns."
[106] → 4	Education	Learning and teaching from code review.
[29] → 1	Code-related aspects	Without explicit definition.
[29] → 2	Share knowledge on the team or project & team	Without explicit definition.
[29] → 3	Sharing knowledge between different seniority levels or roles	Without explicit definition.

In a mixed-method approach, Bacchelli and Bird explored the expectations, outcomes, and challenges of modern code review at Microsoft [7]. From analyzing code review comments and interviews with developers and managers at Microsoft, this seminal study identifies ten different motivations for and expectations towards code review and concludes that although finding defects is a key motivation for code review, only a small portion of the code review comments were defect-related and "mainly cover small, low-level issues". The six motivations listed in Table 3.1 are discussed in detail; the other four motivations are not discussed further in the paper. Not all motivations are explicitly defined and, in our opinion, are not necessarily mutually exclusive. In particular, exploring the relationship between knowledge sharing and the other expectations further was not in the scope of the study. The study thus recommends studying the socio-technical effects of and investigating if and how learning increases as a result of code review.

The study by Baum et al. reports on ten effects (seven desired and three undesired effects) of code review using grounded theory as part of an interview study with 24 software engineering professionals from 19 companies [14]. The study reports seven findings as desired code review effects. Thereby, the study implicitly confirms the reported motivations from [7] although the authors do not discuss the relation to existing evidence explicitly. A frequency or weighting of the findings is not reported and we may, thus, assume an arbitrary ordering. Through the explicit separation between learning for the author and learning for the reviewer, the study also finds a

mutual knowledge transfer, a mutual information exchange, which also supports the notion of bidirectional knowledge transfer as stated in [7].

In two surveys among developers from both an industrial and an open-source context, Bosu et al. contrasted industrial code review at Microsoft with code review at different open-source projects [17]. The primary motivations reported are maintainability, knowledge-sharing, functional defects, community building, minor errors, and others. They found a significant difference in the primary purposes of code review ("RQ 1: Why are code review important") between those two contexts: Open-source developers focus more on knowledge-sharing while developers in open source reported maintainability as a primary expectation towards code review. Eliminating functional defects was only the third most important reason for code reviews in both surveys, which further corroborates the findings by Bacchelli and Bird. However, both studies [7, 17] surveyed the same company: Microsoft. This limits more generalizable conclusions from the authors' findings on our side. Although we also rely on the code review system from Microsoft, we added two further industrial code review systems, Trivago and Spotify. This allows us to broaden our perspective on code review in industry.

In the context of a mixed-methods study, Sadowski et al. conducted an interview study to investigate the motivations for code review at Google [106]. In more detail, the authors conducted interviews with 12 employees working for Google from one month to ten years (with a median of five years). Four key themes emerged: education (learning and teaching from code review), *maintaining norms* (organization preference for a discretionary choice, e.g., formatting or API usage patterns), *gatekeeping* (establishment or maintenance of boundaries around source code, design choices, or other artifacts[1]), and *accident prevention* (reducing introduction of bugs, defects, or other quality-related issues). The authors explain that these expectations can map over those found previously at Microsoft in [7, 17]. Unfortunately, the exact mapping is not presented in the article. The authors emphasize that the main focus at Google, as explained by their participants, is on education as well as code readability and understandability. Why, as stated in the manuscript, this focus contradicts the finding by Rigby and Bird that code review has changed from a defect-finding activity to a group problem-solving activity is not discussed further [99].

Cunha, Conte, and Gadelha report a qualitative survey with 106 practitioners regarding their experiences with modern code review [29]. The paper presents its findings around three codes from the open coding: "code-related aspects", "share knowledge on the team or project", and "share knowledge between different seniority levels and roles". Although details on the practitioners' affiliations are not reported, the surveyed practitioners are affiliated with companies based in Brazil and (in a "smaller" yet unreported proportion) in South Africa, Sweden, Ireland, Spain, and France. This

[1] Upon our request during this study, the authors clarified that gatekeeping refers to requiring a code review from a project owner in order to check in code within a project from someone outside or requiring someone with certification in a particular language to review some code in that language.

broadens the geographical perspective on the expectations towards code review since [14] surveyed companies based in Germany, the Czech Republic, and the USA, [7, 17, 106] report on companies based in the USA (Microsoft and Google).

All five studies have in common that the use of the terms knowledge sharing, transfer, spreading, or learning is neither consistent among (and partially even within) those prior works nor thoroughly defined. This is likely rooted in the complex nature of knowledge and the different epistemological stances. Furthermore, it remains unclear to what extent knowledge transfer differs from all other expectations. For example, knowledge must be transferred when an alternative solution is proposed or a defect is made explicit through a code review.

For the synthesis from prior work on the expectations towards code review, we made the following decisions:

- *Information over knowledge*—We consistently use *information* instead of knowledge for the synthesis of the prior work and throughout this study. We, thereby, concur with [92]. Although not equivalent, information encodes knowledge since knowledge is the meaning that may be derived from information through interpretation. This means that we may see information as a superset of knowledge. Hence, not all information is necessarily knowledge, but all knowledge is information. This allows us to subsume different stances, definitions, and notions of knowledge without an epistemological reflection upon the various definitions of knowledge. Furthermore, we can refrain from delineating the notion of knowledge from the notion of truth, the latter being too often an inherent connotation of knowledge. We may well postulate that not everything communicated is true. Opinions, expectations, misunderstandings, or best guesses are also part of any engineering and development process and do not meet knowledge and, consequently, truth by all definitions.
- *Information exchange, sharing, spreading, or transfer*—We consider information sharing, spreading, or transfer as synonyms for communication, which is the exchange of information. We will discuss and derive our definition of communication, the exchange of information, in Section 3.3.1 in detail.
- *Improvements over benefits*—The term *benefit* implies that there is a positive outcome from a particular action, decision, or situation—from code review. Since code review does not happen in the void, we prefer the term *improvement* to emphasize the context of code review.

All qualitative studies reported the finding that code review is expected to exchange information. In our synthesis, we distinguish between information exchange as the root cause for the expected improvements on the one side and the expected improvements through the information exchange on the other side. We may assume that all reported and expected improvements are caused by the information exchange through code

review: None of the improvements would be possible without exchanging information among the code review participants. Figure 3.1 presents our synthesis of expectations and motivations towards code review reflecting this distinction.

In detail, we found the expected improvements through information exchange in code review either to be related to code or to collaboration. We grouped the findings related to code into functional (identification of defects) and non-functional code improvement. The latter contains three groups of improvements: (1) alternative solutions and their discussions, (2) higher maintainability, and (3) compliance with norms and regulations. The compliance is not limited to regulations (e.g., from regulatory environments such as medical or automotive software development) but also includes organizational norms and practices directed at code. Closely related to the organizational norms and practices is the second group of expected improvements through information exchange in code review, which is collaboration. This also includes team awareness, a sense of shared code ownership, and community building.

3.2.2 Measuring Information Exchange

To the best of our knowledge (or information), there are only two cases so far to quantify knowledge sharing in code review.

The first case of measuring knowledge sharing (or information exchange) in code review was provided by [99]. The authors extended the expertise measure proposed by [78]. The study contrasts the number of files a developer has modified with the number of files the developer knows about (submitted files $\cup$ reviewed files) and found a substantial increase in the number of files a developer knows about exclusively through code review.

The second case of measuring knowledge spreading (or information exchange) is presented by [106], the case study at Google discussed previously. The study reports (a) the number of comments per change a change author receives over tenure at Google and (b) the median number of files edited, reviewed, and both—as suggested by [99]. The study finds that the more senior a code change author is, the fewer code comments he or she gets. The authors "postulate that this decrease in commenting results from reviewers needing to ask fewer questions as they build familiarity with the codebase and corroborates the hypothesis that the educational aspect of code review may pay off over time." In its second measurement, the study reproduces the measurements of [99] but reports it over tenure months at Google. The plot shows that reviewed and edited files are distinct sets to a large degree.

However, we found the following limitations in the measurement applied to prior work:

- We are unaware of empirical evidence that exposure to files in code review would reliably lead to improved developer fluency.

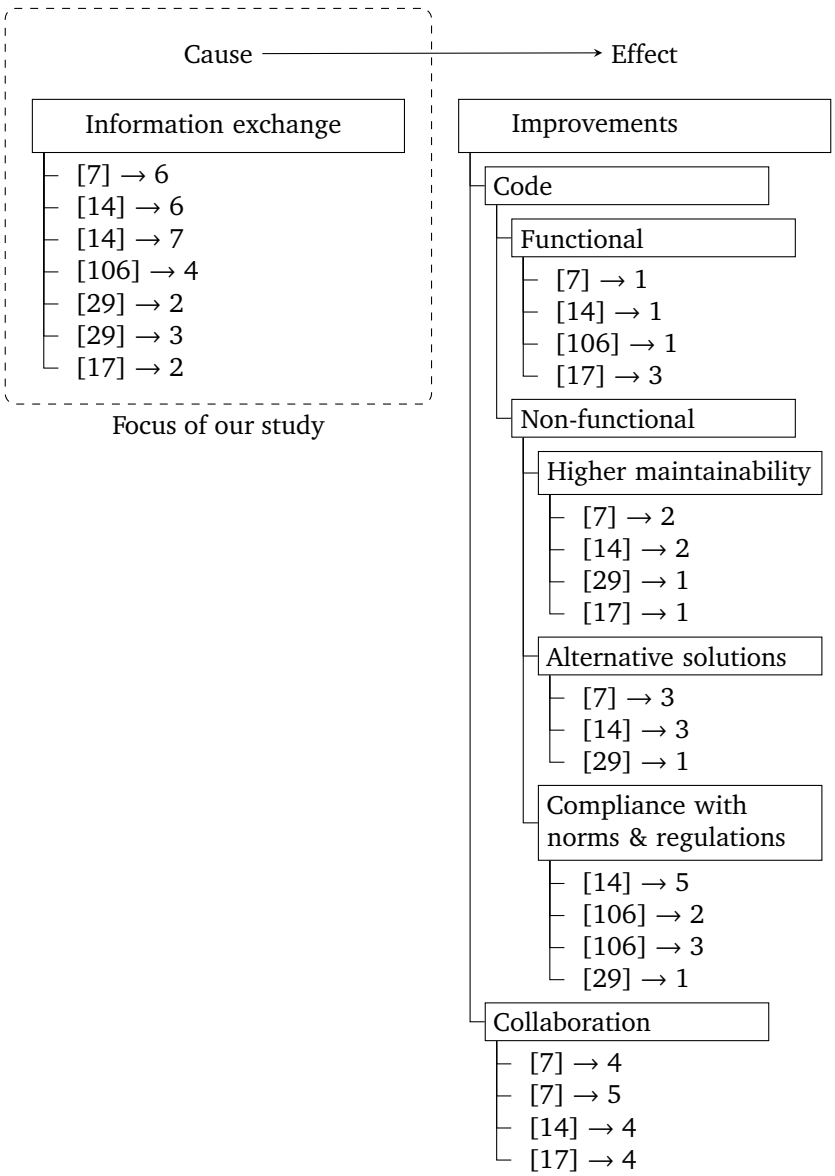


Figure 3.1: Synthesis of expectations towards code review reported by [7, 14, 17, 29, 106]: We consider information exchange the cause for the expected improvements from code review. In this study, we aim to provide a quantitative counterpart on information exchange as qualitatively-reported expectation towards code review.

- Findings like [7] → 4 (team awareness and transparency) and [7] → 5 (share code ownership) cannot be measured.
- The explanatory power of both measurements is limited since the authors set arbitrary boundaries: [99] excluded changes and reviews that contain more than ten files, and [106] limited the tenure to 18 months and aggregated the tenure by three months.

For our approach, therefore, we need to subsume all notions of knowledge by using the broader concept of information and its exchange since information encodes knowledge of all types, including meta-information, such as the social ties between developers. This subsumption was also used in [41] to validate the simulation model, showing the importance of time for measuring and analyzing information diffusion.

3.2.3 Information Diffusion

Information diffusion, the spread of information among humans, has been researched in different disciplines and for encodings of information, for example, tweets [2], memes [71], blog posts [51], or e-mail chain letters [74].

However, information in code review is more fine-grained and significantly harder to identify and trace than forwarded tweets, memes, blog posts, or e-mails. Therefore, [41] proposed and validated a simulation model for information diffusion tailored to code review without tracing identifiable information but focussing on the communication network emerging from code review. In detail, we modelled the code review network emerging from code review as time-varying hypergraph, where the nodes are code review participants and the (hyper)edges are code reviews that connect the participants over time. For more details, we refer the reader to Section 3.3.1 as we reuse the simulation model from [41] in this study.

3.2.4 Simulations as Empirical Research Method

We conduct our study as an *in silico* experiment, in which we simulate an artificial information diffusion and measure the resulting traces generated by the spread of the information. Given the rarity of simulations in the empirical software engineering community, we motivate the explicit choice of that as our (empirical) research method.

Generally speaking, an *in silico* experiment is an experiment performed in a computer simulation (on silicon). In contrast to other types of experiments (see Table 3.2), all parts of the experiment, i.e., the actors, their behaviours, and the context (borrowed from [114]), are modelled explicitly as computer (software) model. In-vitro experiments model the context other than using a computer model.

Those more traditional experiments in software engineering would have been too complex, too expensive, too lengthy, or simply not possible or accessible otherwise.

Table 3.2: A comparison of *in-vivo*, *in-vitro*, *in-virtuo*, and *in silico* experiments.

	Experiment			
	in-vivo	in-vitro	in-virtuo	in silico
Actor	natural	natural	natural	modelled
Behaviour	natural	natural	natural	modelled
Context	natural	modelled	modelled	modelled

otherwise

as computer (software) model

Following [81] "Simulation models are like virtual laboratories where hypotheses about observed problems can be tested, and corrective policies can be experimented with before they are implemented in the real system." Those attributes match the objectives and setting of our research.

Simulation models have been applied in different research fields of software engineering, e.g., process engineering, risk management, and quality assurance [81].

The role of simulations as an empirical method is, however, still often subject to some form of prejudice but also subject to ongoing more philosophical debates. [114], for example, positioned computer simulations in their ABC framework in a non-empirical setting because, as the authors argue: "while variables can be modelled and manipulated based on the rules that are defined within the computer simulation, the researcher does not make any new empirical observations of the behavior of outside actors in a real-world setting (whether these are human participants or systems)" [114]. Without discussing the role of simulations in the empirical software engineering community to the extent they might deserve, however, we still argue for their suitability as an evidence-based (empirical) approach in our context where observations would otherwise not be possible (or, at least, not realistic).

We consider computer simulations as an empirical research method, the same as done in other disciplines and inter-disciplines (where, for instance, climate simulations are the first-class citizens in the set of research methods). Empirical research methods are "research approaches to gather observations and evidence from the real world" [114] and same as in other empirical research methods, in simulation models, we build the models based on real-world observations and make conclusions based on the empirical observations along the execution (in our case, of the simulations). These simulations are abstractions from the real world—same as the (often implicit) theoretical models underlying quasi-controlled (in-vitro) experiments. Simulations

and their underlying models further abstract from (and make explicit) complex systems and make observations and evidence possible in situations where more traditional experiments are rendered infeasible (e.g., too expensive, dangerous, too long, or not accessible) or simply impossible at all; for instance, observations when exploring the capabilities of real-world communication networks with thousands of developers as done in the simulation study presented in the manuscript at hands.

Needless to elaborate, a certain abstraction from the real world is inherent to all empirical research methods, either in the form of explicit models or implicit assumptions. Like every measurement, the models we create come with certain accuracy and precision—with a certain quality. However, we may still argue that the quality of a research method does not necessarily decide upon whether it qualifies as empiricism or not but rather the underlying constructs and their (evidence-based) sources. To avoid surreal models and ensure the quality of a model, however, the modelling itself needs to be guided by quality assurance in verification and validation, and the sample used needs to be realistic; both would, in turn, be in tune with the underlying arguments by available positionings such as the one by [114]. To increase the transparency in the quality of our simulations, we further disclose all developed software components as a replication package, also including the industrial communication networks we used as a sample.

3.3 Experimental Design

In this section, we describe the design of our *in silico* experiment that evaluates and quantifies how far (RQ 1) and how fast (RQ 2) information can diffuse in code review.

The underlying idea of our experiment is simple: We create communication networks emerging from code reviews and then measure the minimal paths between the code review participants. The cardinality of reachable participants indicates how far (RQ 1) information can spread, and distances between participants indicate how fast (RQ 2) information can spread in code review. Since we used minimal paths and created the communication networks under best-case assumptions, the results describe the upper bound of information diffusion in code review.

Yet, since communication, and, therefore, information diffusion, is (1) inherently a time-dependent process that is (2) not necessarily bilateral—often more than two participants exchange information in a code review—, traditional graphs are not capable of rendering information diffusion without dramatically overestimating information diffusion [41]. Therefore, we use time-varying hypergraphs to model the communication network and measure the shortest paths of all vertices within those networks. Since a hypergraph is a generalization of a traditional graph, traditional graph algorithms (i.e., Dijkstra's algorithm) can be used to determine minimal-path distance.

The connotation of minimal is two-fold in time-varying hypergraphs: A distance in time-varying hypergraphs between two vertices has not only a topological but also a temporal perspective. This allows us to measure not only the topologically minimal but also the temporal distance between vertices. Both distance types result in the same set of reachable participants, which we use to answer RQ 1.

Since all models are abstractions and, accordingly, simplifications of reality, the quality of an *in silico* experiment highly depends on the quality of the simulation model and its parametrization. Therefore, we provide a more elaborate description of our simulation model, which was originally proposed and partially validated in [41], in Section 3.3.1 and its parametrization of its computer model by empirical code review data (Section 3.3.2).

3.3.1 Simulation Model

In general, a simulation model consists of two components: (1) the conceptual model describing our derivations and assumptions and (2) the computer model as the implementation of the conceptual model. The following two subsections describe each component in detail.

Conceptual Model

In the following, we describe how we conceptually model the communication networks from code review discussions and the information diffusion within those communication networks.

Communication Network Communication, the purposeful, intentional, and active exchange of information among humans, does not happen in the void. It requires a channel to exchange information. A *communication channel* is a conduit for exchanging information among communication participants. Those channels are

1. *multiplexing*—A channel connects all communication participants sending and receiving information.
2. *reciprocal*—The sender of information also receives information and the receiver also sends information. The information exchange converges. This can be in the form of feedback, queries, or acknowledgments. Pure broadcasting without any form of feedback does not satisfy our definition of communication.
3. *concurrent*—Although a human can only feed into and consume from one channel at a time, multiple concurrent channels are usually used.
4. *time-dependent*—Channels are not available all the time; after the information is transmitted, the channels are closed.

Channels group and structure the information for the communication participants over time and content. Over time, the set of all communication channels forms a communication network among the communication participants.

In the context of our study on information diffusion, a communication channel is a discussion in a merge (or pull) request. A channel for a code review on a merge request begins with the initial submission and ends with the merge in case of an acceptance or a rejection. All participants of the review of the merge request feed information into the channel and, thereby, are connected through this channel and exchange information they communicate. After the code review is completed and the discussion has converged, the channel is closed and archived, and no new information becomes explicit and could emerge. However, a closed channel is usually not deleted but archived and is still available for passive information gathering. We do not intend to model this passive absorption of information from archived channels by retrospection with our model. For this line of research, we recommend the work by [92] as further reading.

From the previous postulates on channel-based communication, we derive our mathematical model: Each communication medium forms an undirected, time-varying hypergraph in which hyperedges represent communication channels. Those hyperedges are available over time and make the hypergraph time-dependent. Additionally, we allow parallel hyperedges[2]—although unlikely, multiple parallel communication channels can emerge between the same participants at the same time but in different contexts.

Such an undirected, time-varying hypergraph reflects all four basic attributes of channel-based communication:

- *multiplexing*—since a single hyperedge connects multiple vertices,
- *concurrent*—since (multi-)hypergraphs allow parallel hyperedges,
- *reciprocal*—since the hypergraph is undirected, information is exchanged in both directions, and
- *time-dependent*—since the hypergraph is time-varying.

In detail, we define the channel-based communication model for information diffusion in an observation window $\mathcal{T}$ to be an undirected time-varying hypergraph

$$\mathcal{H} = (V, \mathcal{E}, \rho, \xi, \psi)$$

where

[2]This makes the hypergraph formally a *multi-hypergraph* [90]. However, we consider the difference between a hypergraph and a multi-hypergraph as marginal since it is grounded in set theory. Sets do not allow multiple instances of the elements. Therefore, instead of a set of hyperedges, we use a multiset of hyperedges that allows multiple instances of the hyperedge.

- V is the set of all human participants in the communication as vertices
- $\mathcal{E}$ is a multiset (parallel edges are permitted) of all communication channels as hyperedges,
- ρ is the *hyperedge presence function* indicating whether a communication channel is active at a given time,
- $\xi\colon E \times \mathcal{T} \to \mathbb{T}$, called *latency function*, indicating the duration to exchange information among communication participants within a communication channel (hyperedge),
- $\psi\colon V \times \mathcal{T} \to \{0, 1\}$, called *vertex presence function*, indicating whether a given vertex is available at a given time.

Information Diffusion The time-respecting routes through the communication network are potential *information diffusion*, the spread of information. To estimate the upper bound of information diffusion and, thereby, answer both of our research questions, we measure the distances between the participants under best-case assumptions.

For information diffusion in code review, we made the following assumptions:

- *Channel-based*—Information can only be exchanged along the information channels that emerged from code review. The information exchange is considered to be completed when the channel is closed.
- *Perfect caching*—All code review participants can remember and cache all information in all code reviews they participate in within the considered time frame.
- *Perfect diffusion*—All participants instantly pass on information at any occasion in all available communication channels in code review.
- *Information diffusion only in code review*—For this simulation, we assume that information gained from discussions in code review diffuses only through code review.
- *Information availability*—To have a common starting point and make the results comparable, the information to be diffused in the network is already available to the participant, which is the origin of the information diffusion process.

Our assumptions make the results of the information diffusion a best-case scenario. Although the assumptions do not likely result in actual, real-world information diffusion, they serve well the scope of our study, namely to quantify the upper bound of information diffusion.

The possible routes through the communication network describe how information can spread through a communication network. Those routes are time-sensitive: a piece of information gained from a communication channel (i.e., a code review discussion) can be shared and exchanged in all subsequent communication channels but not in prior, closed communication channels.

Mathematically, those routes are time-respecting walks, so-called *journeys*, in a time-varying hypergraph representing the communication network. A journey is a sequence of tuples

$$\mathcal{J} = \{(e_1, t_1), (e_2, t_2), \ldots, (e_k, t_k), \}$$

such that $\{e_1, e_2, \ldots, e_k\}$ is a walk in $\mathcal{H}$ with $\rho(e_i, t_i) = 1$ and $t_{i+1} > t_i + \xi(e_i, t_i)$ for all $i < k$.

We define $\mathcal{J}^*_{\mathcal{H}}$ the set of all possible journeys in a time-varying graph $\mathcal{H}$ and $\mathcal{J}^*_{(u,v)} \in \mathcal{J}^*_{\mathcal{H}}$ the journeys between vertices u and v. If $\mathcal{J}^*_{(u,v)} \neq \emptyset$, u can reach v, or in short notation $u \rightsquigarrow v$.[3] Given a vertex u, the set $\{v \in V : u \rightsquigarrow v\}$ is called *horizon* of vertex u.

The notion of length of a journey in time-varying hypergraphs is two-fold: Each journey has a topological distance (measured in number of hops) and temporal distance (measured in time). This gives rise to two distinct definitions of distance in a time-varying graph $\mathcal{H}$:

- The *topological distance* from a vertex u to a vertex v at time t is defined by $d_{u,t}(v) = \min\{|\mathcal{J}(u, v)|_h\}$ where the journey length is $|\mathcal{J}(u, v)|_h = |\{e_1, e_2, \ldots, e_k\}|$. This journey is the *shortest*.
- The *temporal distance* from a vertex u to a vertex v at time t is defined by $\hat{d}_{u,t}(v) = \min\{\psi(e_k) + \xi(e_k) - \xi(e_1)\}$.[4] This journey is the *fastest*.[5]

With this conceptual model and its mathematical background, we are now able to answer both research questions by measuring two characteristics of all possible routes through the communication network:

- The distribution of the horizon of each participant in a communication network represents how far information can spread (RQ 1).
- The distribution of all shortest and fastest journeys between all participants in a communication network answers how fast information can spread in code review (RQ 2). We measure how fast information can spread in code review in terms of the topological distance (minimal number of code reviews required to

[3] In general, journeys are not symmetric and transitive—regardless of whether the hypergraph is directed or undirected: $u \rightsquigarrow v \not\Leftrightarrow v \rightsquigarrow u$.

[4] In our case, $\psi(e_k)$ is always 0.

[5] For the interested reader, we would like to add that if the temporal distance is not defined for a relative time but for an absolute time $\hat{d}_{u,t}(v) = \min\{\psi(e_k) + \xi(e_k)\}$, the journey is called *foremost*. For this line of research, the foremost journeys are not used.

spread information between two code review participants) and the temporal distance (minimal timespan to spread information between two code review participants).

Those measurements within code review communication networks will result in the upper bound of information diffusion in code review.

Computer Model

Since our mathematical model is not trivial and lacks performant tool support for time-varying hypergraphs, we dedicate this section to the computer model and the implementation of the mathematical model described previously.

Time-varying hypergraphs are a novel concept; therefore, we cannot rely on existing toolings. We implemented the time-hypergraph as an equivalent bipartite graph: The hypergraph vertices and hyperedges become two sets of vertices of the bipartite graph. The vertices of those disjoint sets are connected if a hypergraph edge is part of the hyperedge. Figure 3.2 shows a graphical description of the equivalence of hypergraphs and bipartite graphs.

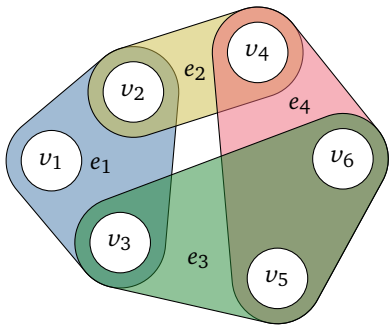
We use a modified Dijkstra's algorithm to find the minimal journeys for each vertex (participant) in the time-varying hypergraph. Dijkstra's algorithm is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. In contrast to its original form, our implementation finds both the shortest (a topological distance) and fastest (a temporal distance) journeys in time-varying hypergraphs.[6]

Since Dijkstra's algorithm can be seen as a generalization of a breadth-first search for unweighted graphs, we can identify not only the minimal paths but also the horizon of each participant in the communication network in one computation.

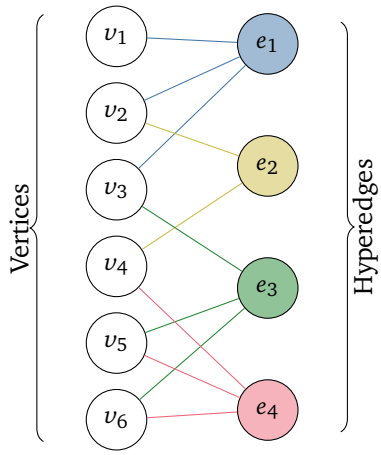
The algorithm is integrated into our computer model and implemented in Python. For more implementation details and performance considerations, we refer the reader to our replication package, including its documentation [33, 34]. Because both time-varying hypergraphs as the data model and the extended Dijkstra's algorithm are novel, we ensure the computational model accurately represents the underlying mathematical model and the correctness of our Dijkstra implementation and its results through the following quality assurance measures:

- *Code walk-throughs*—We independently conducted code walk-throughs through the simulation code with three Python and graph experts.
- *Comprehensive test setup*—The simulation code has a test coverage of over 99 %.

[6] For future applications, our implementation of Dijkstra's algorithm can also find any foremost journey.



(a) An example time-varying hypergraph whose so-called hyperedges (denoted by $e_\square$) can link any arbitrary number of vertices (denoted by $v_\square$): For example, hyperedge e_3 connects three vertices. The horizon and minimal paths of vertex depend highly on the temporal order of the hyperedges: for example, the horizon of v_1 contains all vertices if the temporal availabilities of the hyperedges are $e_1 < e_2 < e_4 < e_3$, but none if $e_1 > e_2 \geq e_3$.



(b) Any hypergraph can be transformed into an equivalent bipartite graph: The hyperedges and the vertices from the time-varying hypergraph from (a) become the two distinct sets of vertices of a bipartite graph.

Figure 3.2: An example hypergraph (a) and its bipartite-graph equivalent (b).

- *Code readability and documentation*—We provide comprehensive documentation on the usage and design decisions to enable broad use and further development. We followed the standard Python programming style guidelines PEP8 to ensure consistency and readability.
- *Publicly available and open source*—The model parameterization and simulation code [34] as well as the results [33] for replications and reproductions are publicly available.

3.3.2 Model Parametrization

Instead of a theoretical or probabilistic parametrization, we parametrize our simulation model with empirical code review systems from three industrial partners: Microsoft, Spotify, and Trivago.

In the following, we describe our sampling strategy for selecting suitable code review systems in industry and the code review data extraction for parametrizing our simulation model.

Sampling

Communication networks do not emerge in the void. They form around software development. As motivated in the introduction, we focus in our study on industrial software development since we believe that results found in open-source projects cannot be directly transferred due to the differences in governance structures, incentives, and economic mechanics. Also, previous qualitative work, which we aim to complement with our work (see Section 3.2.1), considers industrial software development only.

We use a *maximum variation sampling* to select suitable code review communication networks in an industrial context. A maximum (or maximum heterogeneity) variation sampling is a non-probabilistic, purposive sampling technique that chooses a sample to maximize the range of perspectives investigated in the study in order to identify important shared patterns that cut across cases and derive their significance from having emerged out of heterogeneity [117].

We aim for representativeness on two specific dimensions:

- *Code review system size*—To avoid a bias introduced by network effects, we required communication networks emerging from different sizes of code review. The size of a communication network can be measured in terms of the number (hyper)edges (corresponding to the number of code reviews) or vertices (corresponding to the number of participants). In our sample, we use a small (Trivago), mid-sized (Spotify), and large (Microsoft) code review system (see Table 3.3). The size classification in small, mid-sized, and large code review

Table 3.3: Our sample of code review systems with respect to the two dimensions of representativeness: code review system size and tooling.

	Code review system size			Tooling
	Classification	Code reviews	Participants	
Trivago	small	2442	364	BitBucket
Spotify	mid-sized	22 504	1730	GitHub
Microsoft	large	309 740	37 103	CodeFlow

systems is arguably arbitrary and relative to the code review systems in our sample rather than following a general norm that, to the best of our knowledge, does not exist.

- *Code review tool*—In particular, since [14] suggested code review tool in use as a main factor shaping code review in industry, we aim to minimize the code review tool bias for the results and require our sample to contain a diverse set of code review tools. Our sample contains three different code review tools: BitBucket, GitHub, and CodeFlow.

In alignment with the qualitative prior work, we explicitly excluded the different manifestations in code review practices as a sampling dimension. To ensure that the results are comparable within and among the selected contexts and to ease the data extraction, we restrict our population to having a single, central code review tool in use. This means our population is any industrial software development company with a single, centralized code review tool. Like any purposive sampling technique, the maximum variation sampling does not require a sampling frame [9].

From this population, we drew a sample of three industrial cases: Microsoft, Spotify, and Trivago. Table 3.3 provides an overview of our sample of code review systems and the dimension of representativeness. We describe the cases in our sample in more detail in the following subsections.

Microsoft Microsoft is a multinational enterprise that produces computer software, consumer electronics, personal computers, and related services and is based in the USA. We extracted the data from Microsoft's internal code review tool *CodeFlow* [18] run by Azure DevOps service. Although not Microsoft's only code review tool, it covers the vast majority of the company's code review activity. The dataset contains 37 103 code review participants and 309 740 code reviews within the observation window between 2020-02-03 and 2020-03-02.

Spotify Spotify is a multinational enterprise based in Sweden that develops a multimedia streaming platform. We extracted all internal pull requests and their related

	Observation window (four weeks)	Collected during
Microsoft	2020-02-03 to 2020-03-02	May 2020
Spotify	2020-02-03 to 2020-03-02	March 2023
Trivago	2019-11-04 to 2019-12-01	December 2022

Table 3.4: Observation window and the data collection timeframe among our cases.

comments within the observation window between 2020-02-03 and 2020-03-02 from Spotify's GitHub Enterprise instance, the central tool for software development at Spotify. The dataset contains 1730 code review participants and 22 504 code reviews.

Trivago Trivago is a German company developing an accommodation search engine. As a code review tool, Trivago used Bitbucket during the observation window between 2019-11-04 and 2019-12-01. The dataset contains 364 code review participants and 2442 code reviews.

Data Collection

We extract all human interactions with the code review discussions within four consecutive calendar weeks from the single, central code review tools in each industrial context.

We define a code review interaction as any non-trivial contribution to the code review discussion: Creating, editing, approving or closing, and commenting on a code review. For this study, we do not consider other (tool-specific) types of discussion contributions, for example, emojis or likes to a contribution to a code review.

The beginning and end of those four-week timeframes differ and are arbitrary, but share the common attributes: All timeframes

- start on a Monday and end on a Sunday,
- have no significant discontinuities by public holidays such as Christmas,
- are pre-pandemic to avoid introduced noise from introduced work-from-home policies, pandemic-related restrictions, or interferences in the software development.

Table 3.4 lists the timeframes (each four weeks) and when the data was collected.

All non-human code-review participants and interactions (i.e., bots or automated tasks contributing to the code-review discussions) are excluded. We strictly anonymized all participants and removed all identifiable personal information to protect the privacy of all individuals.

All data and results are publicly available [33].

3.4 Results

This section presents the results of our simulation as described in Section 3.3 and is structured around our two research questions.

Both research questions cover different perspectives on code review as a communication network: In RQ 1, we use a vertex-centric perspective by measuring the reachable participants for each participant (vertex). For RQ 2, we use a hyperedge-centric perspective by measuring the topological and temporal lengths of paths through the communication network that emerges from code review.

3.4.1 How far can information diffuse through code review (RQ 1)?

As described in Section 3.3.1, we answer RQ 1 by measuring the number of reachable participants for each participant in the communication network that emerges from code review. The number of reachable participants is the cardinality of each participant's horizon. In the following, we call the number of reachable participants *information diffusion range*.

To make the different code review system sizes comparable, we normalize the information diffusion range to the number of code review participants in a code review system. Mathematically, we define the normalized information diffusion range for all code review participants $u \in V$ by

$$\frac{|\{v \in V : u \rightsquigarrow v\}|}{|V|}.$$

Figure 3.3 plots the empirical cumulative distribution functions (ECDF) visualizing the distributions of the information diffusion range per participant after four weeks each resulting from our simulation.

We found the upper bound of the relative information diffusion range at Trivago's code review system. In detail, a code review participants at Trivago can reach 85 % of its network at maximum. 30 % of the code review participants can reach between 81 % and 85 %, while an average (median) code review participant can reach between 72 % and 85 % of all participants within the network. The code review system at Spotify generates an almost identical distribution of reachable participants. Table 3.5 lists the ranges of horizons possible for the p-percentiles 0.7, 0.5, 0.3 and 0.1.

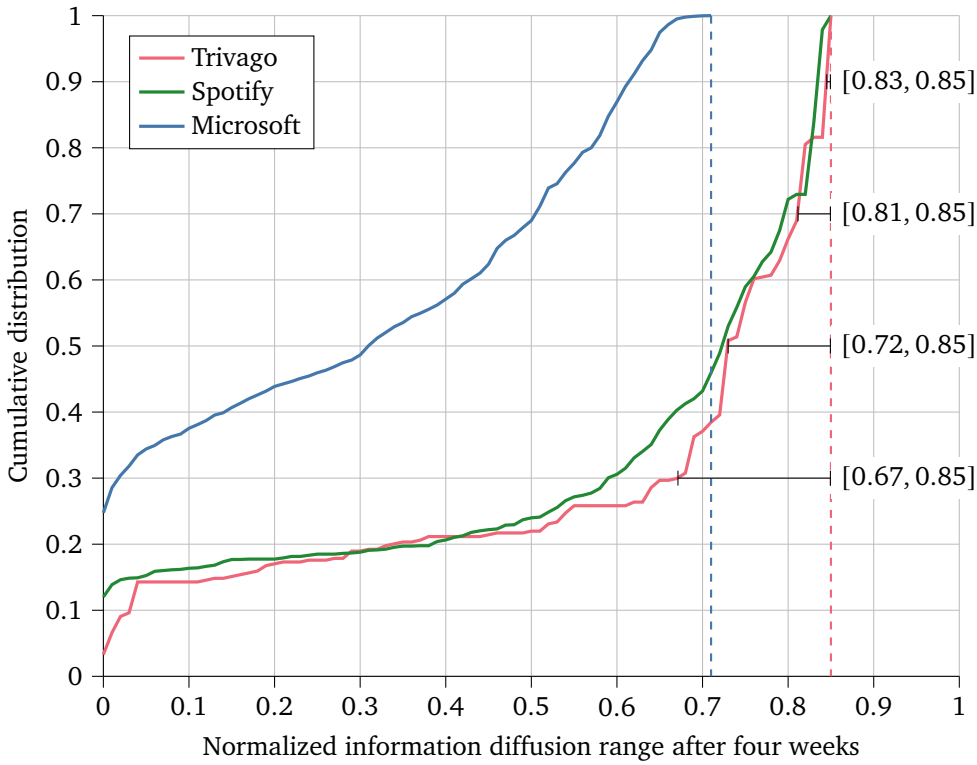


Figure 3.3: Cumulative distribution of information diffusion range per participant. The smallest y value for a given x among all three distributions indicates the upper bound of information diffusion with respect to how many participants can be reached (RQ 1). For example, 30 % of all participants at Trivago can reach between 0 % and 67 % and 70 % of the participants reach between 67 % and 85 % of all participants.

Table 3.5: Cumulative distribution of normalized information diffusion range per participant for the percentiles 0.7, 0.5, 0.3 and 0.1, and the maximum for each code review system.

	0.70	0.50	0.30	0.10	max
Trivago	0.67 to 0.85	0.72 to 0.85	0.81 to 0.85	0.83 to 0.85	0.85
Spotify	0.58 to 0.85	0.72 to 0.85	0.79 to 0.85	0.83 to 0.85	0.85
Microsoft	0.01 to 0.71	0.30 to 0.71	0.50 to 0.71	0.61 to 0.71	0.71

Simulation Result 1

The code review networks at Trivago and Spotify describe almost identically the upper bound of the normalized information diffusion range: Half of the participants at Trivago and Spotify can reach between 72 % and 85 % of all participants within four weeks under best-case assumptions.

↗ Figure 3.3

If we consider the absolute information diffusion range for each code instead review participant $u \in V$ defined by

$$|\{v \in V : u \rightsquigarrow v\}|,$$

code review participants at Microsoft's code review system can reach by far the most participants. Although the relative information diffusion range at Microsoft's code review system is significantly smaller, Microsoft's code review system sets the upper bound for the absolute information diffusion range. In detail, the code review system at Microsoft can spread information up to 26 216 participants (71 % of the total network size), half of the code review participants can reach 11 645 or more other participants. Table 3.6 lists the ranges of the top percentiles.

Simulation Result 2

The code review network at Microsoft describes the upper bound of the absolute information diffusion range: Half of the participants at Microsoft can reach between 11 645 and 26 216 participants within four weeks under best-case assumptions.

↗ Table 3.6

3.4.2 How fast can information diffuse through code review (RQ 2)?

As described in Section 3.3.1, we answer RQ 2 by measuring the distances between the code review participants. We recall that the notion of distance in time-varying hypergraphs is two-fold: Each time-respecting path (journey) has a topological distance (the minimal number of hops of all journeys) and temporal distance (measured in time of all journeys).

Therefore, we align the answers to RQ 2 with those two types of distances.

Topological Distances in Code Review

Figure 3.4 depicts the cumulative distribution of topological distances between code review participants among the sampled cases.

Table 3.6: Cumulative distribution of information diffusion range per participant for the percentiles 0.7, 0.5, 0.3 and 0.1, and the maximum for each code review system.

	0.70	0.50	0.30	0.10	max
Trivago	245 to 310	266 to 310	296 to 310	309 to 310	310
Spotify	1026 to 1472	1260 to 1472	1386 to 1472	1447 to 1472	1472
Microsoft	808 to 26 216	11 645 to 26 216	18 887 to 26 216	22 983 to 26 216	26 216

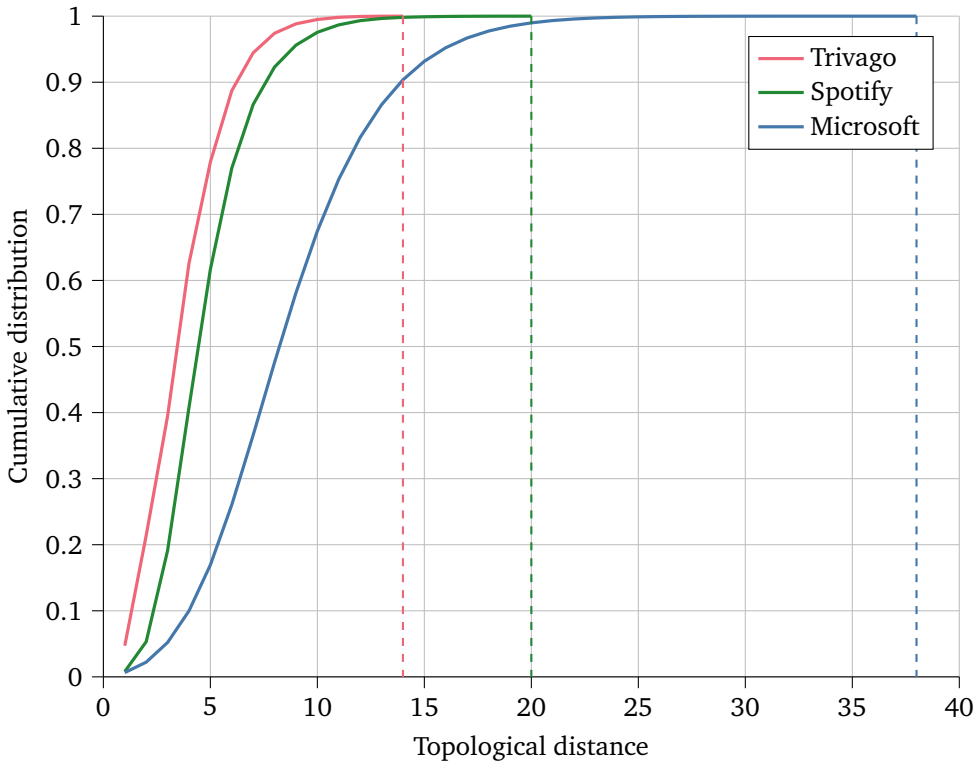


Figure 3.4: The cumulative distribution of topological distances between participants in code review systems. The topological distance is the minimal number of code reviews (hops) required to spread information from one code review participant to another.

The code review system at Trivago contains the most distances among our three cases. The average (median) distance between two participants at Trivago is three, at Spotify four, and at Microsoft eight hops. 60 % of all distances at Trivago and Spotify are shorter or equal to five code reviews. The maximum distance per case is 14 for Trivago, 20 for Spotify, and 38 for Microsoft.

Simulation Result 3

Trivago's code review system describes the upper bound on how fast information can spread through code review: About 75 % of all distances in Trivago's code review system between code review participants are shorter than five code reviews.

↗ Figure 3.4

Temporal Distances in Code Review

The other type of distance in time-varying hypergraphs is the temporal distance. The fastest time-varying path is the path between two code review participants with the minimal (relative) temporal distance between two code review participants describes the minimal timespan to spread information from one participant to another. Due to our observation window, the temporal distance cannot exceed four weeks in our measurement. Figure 3.5 depicts the cumulative distribution of the relative temporal distances between the code review participants in our sample.

The average (median) temporal distance between two code review participants at Trivago or Spotify is less than seven days, while a code review participant a Microsoft takes more than 14 days, which is still in the observation window of four weeks.

Simulation Result 4

Trivago's code review system describes the upper bound on how fast information can spread through code review concerning the relative temporal distance: The average (median) temporal distance between two code review participants at Trivago are five days.

↗ Figure 3.5

3.5 Discussion

Both research questions cover two different and complementary perspectives on communication networks that emerge from code review. RQ 1 captures a vertex-centric perspective on code review focusing on the participants as nodes in the communication network and their horizon. RQ 2 captures a (hyper)edge-centric perspective focussing on the length of minimal paths through the networks, representing the code

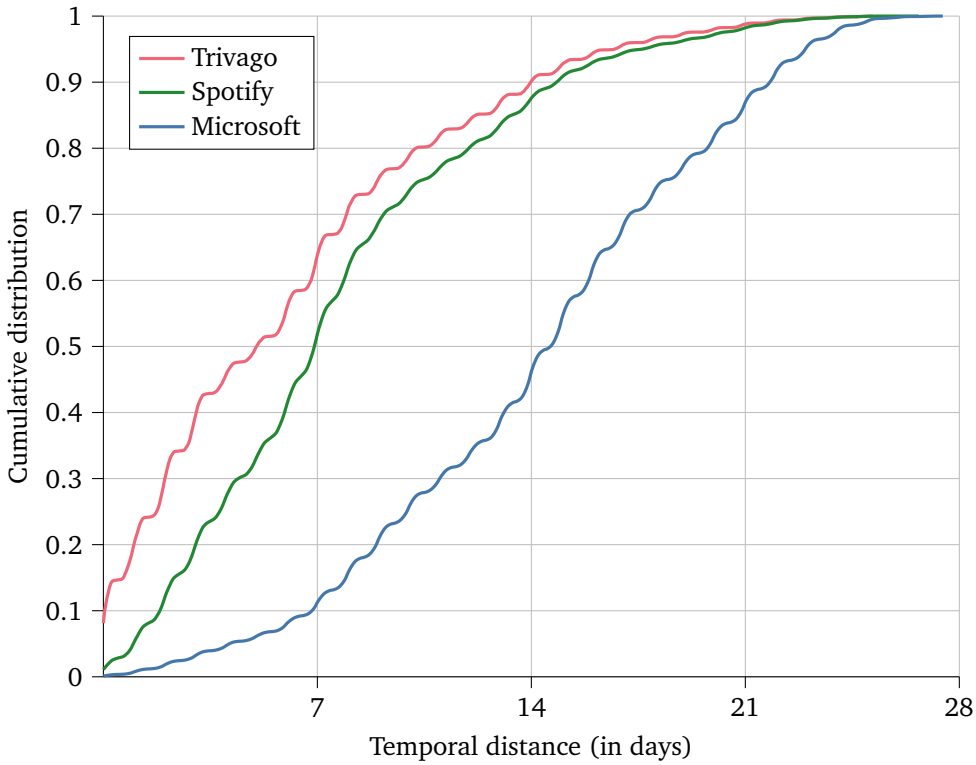


Figure 3.5: The cumulative distribution of minimal temporal distances between participants in code review systems. The temporal distance is the minimal duration required to spread information from one participant to another.

reviews as communication channels connecting the participants. Also, for this section, we group our discussion around the two research questions.

3.5.1 How far can information diffuse through code review (RQ 1)?

We found a relative upper bound on how far information can spread through code review for small and mid-sized code review systems and an absolute upper bound for large code review systems (see Results 1 and 2).

Although the code review system at Trivago describes the upper bound on how far information can spread through code review (RQ 1), the code review system at Spotify has almost identical distributions of normalized information diffusion range—despite a different tooling and different total network size. Both code review systems define the upper bound of how far information can spread relative to the network size (see Result 1). We consider that as a first indicator that the choice of tooling is secondary. However, although the similarity between the small and mid-sized code review systems is striking, this study was not designed to examine patterns among the code review systems. Thus, we cannot exclude a random correlation.

Microsoft's code review system, as the largest code review system, however, describes the absolute upper bound on how far information can spread (see Result 2).

We are surprised by those two results as we expected a significantly smaller relative and absolute upper bound that is more guided by the organizational or software architectural boundaries. Although neither organizational nor software architectural information is available for this study, we assume an information diffusion beyond organizational or software architectural boundaries among all cases because all information diffusion ranges are magnitudes larger than reasonable team sizes. Although this study does not evaluate the expected improvements of code review but focuses on the underlying expected cause of information exchange through code review (see Figure 3.1), this finding corroborates the expectation [7] → 4 towards team awareness and transparency: Information can leave the organizational boundaries in code review leading to improved collaboration.

3.5.2 How fast can information diffuse through code review (RQ 2)?

Although we found both the topological and temporal distances to be minimal in Trivago's code review system (see Results 3 and 4), the temporal distances we measured at Spotify and Microsoft are remarkable given their code review sizes on almost a logarithmic scale: The average (median) distance at Spotify's code review system is shorter than five code reviews (topological distance) and shorter than seven days (temporal distance); The average (median) distance at Microsoft's code review system is shorter than seven code reviews (topological distance) and shorter than about 14 days (temporal distance).

The step-like characteristics among all cases, but more prominent in Trivago's and Microsoft's code review system, indicate a common day-night and workday rhythm of the two participants connected by the fastest time-respecting path. Although the developers' locations are not available to us and the investigation is out of the scope of our study, we speculate that information diffusion in the code review systems at Trivago and Microsoft stays mostly in the same timezone. However, Spotify's code review system describes a less distinct pattern with less clear steps. Therefore, we assume an information diffusion beyond timezones at Spotify's code review system.

3.6 Limitations

In outlining the limitations of our study, it is crucial to acknowledge potential threats to validity. This section highlights key constraints, both internal and external, providing context for the interpretation of our findings and suggesting avenues for future research.

3.6.1 Best-Case Assumptions

As already discussed in Section 3.3.1, our assumptions regarding the information diffusion make the results a best-case scenario that is unlikely to be achieved in reality: Information is unlikely to spread on every occasion or to all code review participants. Information diffusion depends on the capability of human participants to buffer, filter, and consolidate information from their minds. Since we are unaware of any prior work on those capabilities, the results remain a theoretical upper bound of information diffusion but are no information diffusion in code review.

3.6.2 Non-human Code Review Activities

Although we excluded all code review bots from the network, the effects of bot activities on the communication network still remain:

First, we found evidence that participants (at least partially) automated the code review handling. Bots disguised as human participants can distort the results since those bots connect more code reviews and, therefore, people than humans do. After removing all known and explicitly labeled bots, we found 14 accounts that contributed to more than 500 code reviews during our observation window of four weeks. All of those were in Microsoft's code review systems. Assuming 20 workdays within our observation windows and 8 hours a day, 500 code reviews within four weeks means about three code reviews per hour on average (mean). The maximum number of contributed code reviews is 8249 which then corresponds to about 50 code reviews per hour on average (mean). We consider that code review load is possible but highly unlikely. We did not remove the questionable accounts for the following reasons:

- We are unaware of empirical studies reporting the upper bound of a code review load in industry. Existing prior work on workload-aware code review participant selection does not report a distribution of code review involvement, normalize the code review size to the number of involved files, or is based on open source projects [4, 25, 116, 137]. Any threshold would be, therefore, arbitrary.
- We cannot distinguish between bot activity (for example, a one-time cleaning script of the code review) and an actual human within such a questionable account.

An in-depth inspection was not possible as it would require a complete de-anonymization of the accounts and analysis of the content of the code reviews of those which is not covered by the study's non-disclosure agreement.

Second, bots can provide assistance for or enforce code review guidelines by selecting and informing a set of code review participants. Those bot activities shape the network drastically and are not covered by our work.

In the foreseeable future, LLM-based bots may become code review participants. On the one hand, they produce code that is required to be reviewed by a human as long as machines cannot be held liable and accountable and they can provide feedback and share important contextual information. However, these increased bot activities may increase the workload of the human reviewers and even slow down the communication through code review [126]. The promising work by [105] can lay the foundation for understanding the impact of bots on communication, coordination, and collaboration.

Thus, excluding those code review activities would not reflect the information diffusion in code review anymore in the near future. However, since all the observation windows for all code review systems were located before the rise of LLMs, we are convinced that excluding bot accounts is appropriate. Future work is needed to investigate the impact of non-human code review from a communication network perspective.

3.6.3 Observation Window

As for any continuous, real-world process, we only can make assumptions about windowed observations of that phenomenon. At the border of our observation windows, we have to live with some blur and uncertainty: The communication channel may have started before or ended after our observed time window. Figure 3.6 demonstrates the problem of the observation window for an ongoing system. A channel is either

- unbounded (observation window does not include start or end of the channel)
- bounded (observation window contains start and end of the channel)
- left-bounded (observation window contains start but no end of the channel)

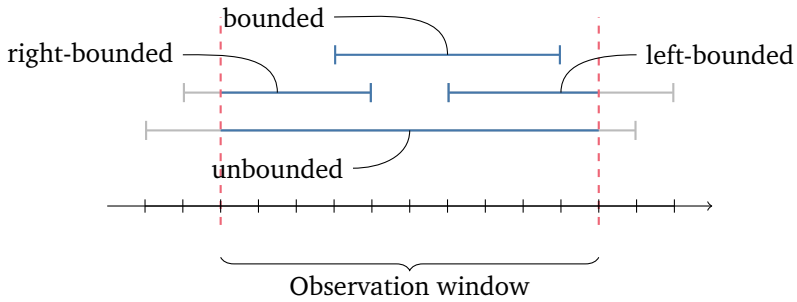


Figure 3.6: The impact of observation windows on data completeness: The concurrent code reviews as communication channels may have started before or ended after the observed time window. Due to the cut, the communication channels may cut at their start (right-bound) or at their end (left-bound), or the channel is completely contained (bound) or not contained (unbound) in the observation window.

- right-bounded (observation window contains end but not start of the channel)

In particular, in communication channels (code reviews) that are right-bounded or unbound, we may miss participants who contributed to the discussion and, therefore, can spread information. Left-bounded and bounded communication channels do not contribute to this uncertainty since we know all participants within the observation window. In Figure 3.7, we see the distributions of the communications-channel bounds.

Although the observation windows of four weeks are arbitrary and, therefore, all distances longer than four weeks are not captured, we consider the observation window as sufficiently large enough to capture the information diffusion through code review: All code review networks reached a plateau regarding how far information can spread through code review. Figure 3.8 is a comprehensive overview of our simulation results depicting the distributions of reachable code review participants over time as color-coded inter-percentile ranges for all three code review systems.

We observe that the code review systems at Trivago and Spotify reach a plateau after two, and the code review system at Microsoft after three weeks on how far information can spread. That means a larger observation window and, therefore, longer topological and temporal distances would likely not significantly impact the information diffusion and the number of reachable participants.

Not only the size of the observation window but also its positioning in time can affect our results. Larger discontinuities (e.g., holidays for large parts of the staff, vacation seasons over summer) and external interferences (e.g., the pandemic, large-scale outages of the development infrastructure, etc.) with the software development will affect code review, and, thus, impact our results. We void such noise that would have impacted our results by carefully selecting pre-pandemic observation windows with

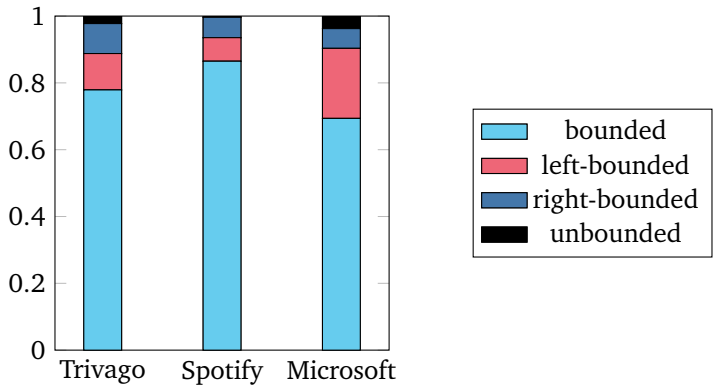


Figure 3.7: Share of bounded, left-bounded, right-bounded, and unbounded communication channels among all cases.

no significant discontinuities by public holidays such as Christmas (see Section 3.3.2). Therefore, we believe that the positioning of the timeframes in time for code review as a continuous endeavor has no significant influence on the results. However, we are not able to provide empirical evidence to validate this claim (i.e., the observation window covers a typical, presentative month) beyond the assessment of our industry partners when selecting the timeframe.

3.6.4 Generalization

The generalization of our results highly depends on our sampling strategy. In general, our study is affected by an availability bias: Companies are hesitant to share code review data since the code review system—as any communication tool—may contain confidential and personally identifiable information of their developers. However, we used a maximum variation sampling to select suitable code review systems in an industrial context and aimed for representativeness on code review system size and tooling.

The size classification of code review systems is arbitrary and relative to our sample. On the lower bound, we classified a code review system with 364 participants to be small, although there are code review systems that are significantly smaller. On the upper bound, however, our sample includes arguably one of the largest code review systems nowadays with more than 37 103 participants.

Our sampling contains three code review tools: GitHub (via pull/merge requests) at Spotify, BitBucket (via merge requests) at Trivago, and CodeFlow as an internally developed tool at Microsoft. This, however, does not cover the tool landscape extensively: In particular, we miss *Gerrit*, *Phabricator*, and *Gitlab* from the broadly available

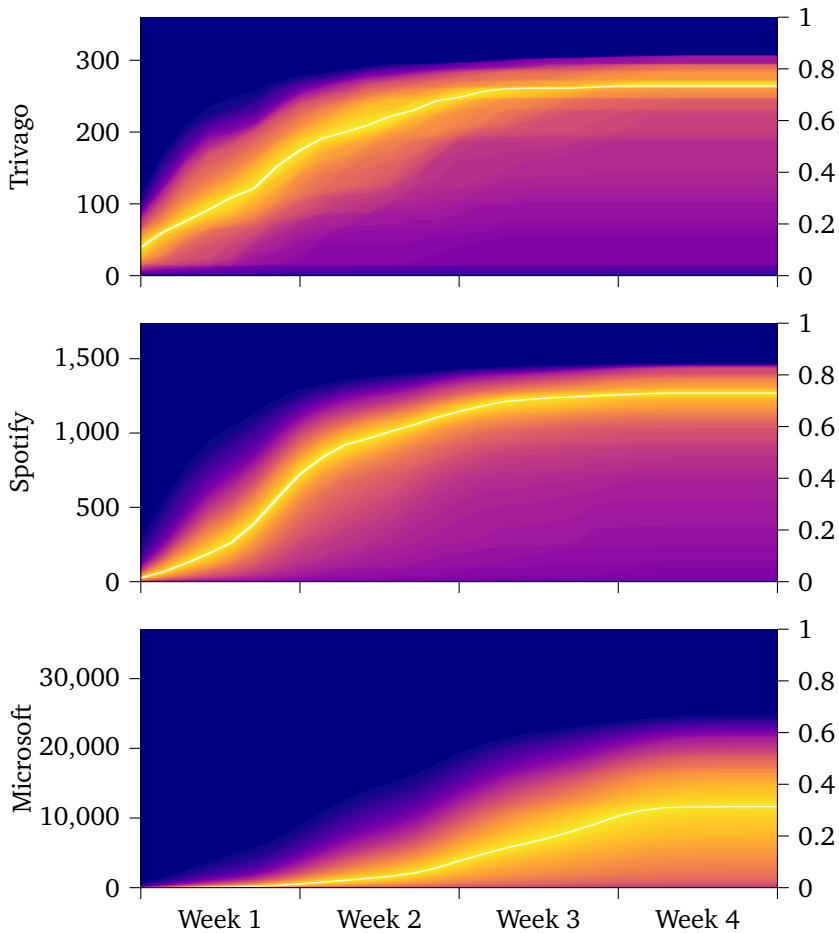


Figure 3.8: The information diffusion range absolute (left y-axis) and relative to the network size (right y-axis) distribution per participant in the observation window of four weeks (x-axis). The distribution is presented as a color-coded intra-percentile range with the median in white.

tools. We believe that missing Phabricator and Gitlab does not introduce a bias: The future of Phabricator is unknown since the company running the developed, *Phacility*, ended all operations [96] and the code review (via merge requests) in Gitlab is, in our opinion, equivalent to that in GitHub. However, Gerrit, as a popular and dedicated code review tool with its voting system, could lead to other communication networks and, therefore, to different information diffusion results.

We explicitly excluded the practices as sampling dimensions although they have a direct impact on the resulting communication network: Code review guidelines, code ownership, or other selection criteria define who is invited to and has a say in a code review discussion and, therefore, prescribe and limit the available communication channels for the information sharing. Those guidelines vary among companies but also within them.

3.7 Conclusion

With this study, we make a first quantitative step towards understanding and evaluating code review as a communication network at scale. Through our information diffusion simulation based on communication networks emerging from code review systems at Microsoft, Spotify, and Trivago, we found an upper bound of information diffusion in code review:

On average (median), information can spread between 72 % and 85 % of all participants for small and mid-sized code review systems or between 30 % and 71 % of all participants for large code review systems (Microsoft); which corresponds to an absolute range between 11 645 and 26 216 code review participants. This describes the upper bound on how far information can spread in code review

The average (median) distance between two code review participants is shorter than three code reviews or five days at small code review systems (Trivago). The average distance in mid-sized (Spotify) and large (Microsoft) code review systems ranges between four and seven code reviews or seven and fourteen days—considering the sizes on an exponential-like scale a significant finding. This describes the upper bound on how fast information can spread in code review.

Our findings align with findings from the prior qualitative research: All five relevant prior studies reported information exchange among code review participants as a key expectation towards code review. Our findings (indicating that the communication network that emerges from code review is capable of diffusing information fast and far) corroborate the qualitatively reported information exchange as an expectation towards code review, which we consider the foundation for all other reported and expected benefits of code review. Although we argue that given the sheer magnitude of information diffusion possible through code, information must cross organizational boundaries and, therefore, code review enables collaboration in companies at scale,

future work is required to investigate and establish a thorough connection between our work and an improved collaboration at scale, another qualitatively reported expectation towards code review. This applies to code-related expected improvements, too.

Although our sample of code review systems limits the generalizability of our findings, we conclude for researchers and practitioners alike:

- *The larger the better.* Because code review is a communication network that can scale with the information diffusion among its participants, companies may unify and centralize their code review systems—independently of (monolithic) code repositories [95].
- *Tooling is secondary.* We did not find any evidence that the choice of tooling plays a crucial role in information diffusion through code review.
- *The role of bots rethought.* Although bots can provide assistance for selecting and informing a set of code review participants optimal with respect to information diffusion, bots tend to introduce noise in the communication channels [126] that may slow down the communication through code review. The promising work by Röseler, Scholtes, and Gote can lay the foundation for understanding the impact of bots on communication, coordination, and collaboration.

Our comprehensive replication package enables researchers to fully reproduce our results and replicate our study using other code review systems to parametrize our simulation model.

The need for replication applies in particular to open source. Not only because [17] already reported a significantly large difference in the primary motivation for code review in an industrial and open-source context, we believe that findings from an industrial or open-source context are not easily transferrable: The mechanics and incentives in open source differ, and so do the organizational structure, liability, and commitment [11].

We also invite to enhance our simulation. In particular, implementing diffusion probabilities could broaden our understanding of information diffusion beyond the best-case assumptions.

So far, we have explicitly excluded the underlying code review practices as a sampling dimension. These practices most likely significantly impact the resulting communication networks emerging from code review and, therefore, the information diffusion in code review. Future work could map practices to information diffusion to indicate the reasonable cost-benefit ratio of code review.

In this study, we focused on the upper bound of information diffusion and answered the research questions regarding how far and how fast information can spread through code review. Our research design does not aim to investigate how fast and how far

information actually spreads through code review; it remains an estimation of the upper bound of information diffusion in code review. In future work, we will measure (rather than simulate) the actual information diffusion through code review. Therefore, we will develop a measurement system to follow the traces in the communication networks that emerge from code review. Code review tools like GitHub provide a foundation for those investigations.

Acknowledgement

We thank Andreas Bauer for his valuable feedback on the technical aspects of the simulation. We are very grateful for the support from our industry partners, in particular, from Andy Grunwald, Simon Brüggen, and Marcin Floryan. We also thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and helpful suggestions, and Daniel Graziotin from the Open Science Board at EMSE for his prompt and profound feedback on our replication package. This work was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

Estimating Information Diffusion in Open-Source Code Review

4

This chapter is based on Michael Dorner and Daniel Mendez. "*The Capability of Code Review as a Communication Network*". Manuscript currently under review at *Journal of Empirical Software Engineering*. 2025.

Abstract

Background: Code review, a core practice in software engineering, has been widely studied as a collaborative process, with prior work suggesting it functions as a communication network. However, this theory remains untested, limiting its practical and theoretical significance.

Objective: This study aims to (1) formalize the theory of code review as a communication network explicit and (2) empirically test its validity by quantifying how widely and how quickly information can spread in code review.

Method: We replicate an *in silico* experiment simulating information diffusion—the spread of information among participants—under best-case conditions across three open-source (Android, Visual Studio Code, React) and three closed-source code review systems (Microsoft, Spotify, Trivago) each modelled as a communication network. By measuring the number of reachable participants and the minimal topological and temporal distances, we quantify how widely and how quickly information can spread through code review.

Results: We demonstrate that code review can enable both wide and fast information diffusion, even at a large scale. However, this capacity varies: open-source code review spreads information faster, while closed-source review reaches more participants.

Conclusion: Our findings reinforce and refine the theory, highlighting implications for measuring collaboration, generalizing open-source studies, and the role of AI in shaping future code review.

4.1 Introduction

Modern software systems are often too large, too complex, and evolve too fast for an individual developer to oversee all parts of the software and, thus, to understand all implications of a change. Therefore, most software projects rely on code review to foster informal and asynchronous discussions on changes and their impacts before they are merged into the codebases. During those discussions, participants exchange information about the proposed changes and their implications, forming a communication network that emerges through code review.

This perspective on code review as a communication network is supported by a broad body of prior qualitative research. As a core practice in collaborative software engineering, the communicative and collaborative nature of code review has been examined in various studies [7, 14, 17, 29, 106]. Our synthesis of this prior research revealed a consistent pattern: practitioners widely perceive code review as a communication medium for information exchange, effectively functioning as a communication network [39]. This recurring pattern provides a solid foundation for the emerging

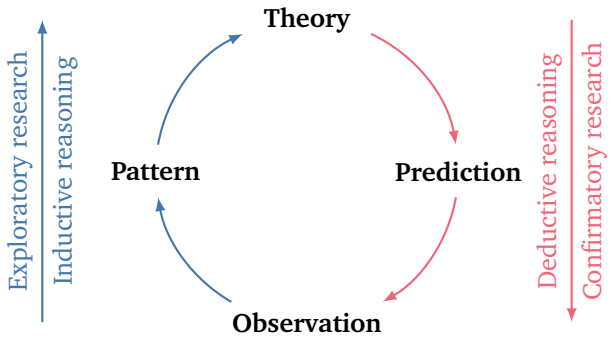


Figure 4.1: The empirical research cycle (in analogy to [77]): While exploratory research is theory-generating using inductive reasoning (starting with observations), confirmatory research is theory-testing using deductive reasoning (starting with a theory). This research is confirmatory.

theory of code review as a communication network, which conceptualizes code review as a process that enables participants to exchange information around a code change.

However, the theory—as often in software engineering research [115]—remains implicit and has yet to be formalized. Furthermore, relying solely on this mostly exploratory and qualitative research to ground our understanding of code review as a communication network falls short. Exploratory research begins with specific observations, distills patterns in those observations, typically in the form of propositions or hypotheses, and derives theories from the observed patterns through (inductive) reasoning. The nature of exploratory and especially qualitative research enables to analyze chosen cases and their contexts in great detail. However, their level of generalizability is also limited as they are drawn from those specific cases (especially when there is little to no relation to existing evidence that would allow for generalization by analogy). Therefore, while such initial theories may well serve as a very valuable foundation, they still require multiple confirmatory evaluation" as, otherwise, their robustness (and scope) remains uncertain rendering it difficult to establish credibility, apply in practice, or develop it further. We thus postulate that exploratory (inductive) research alone is not sufficient to achieve a strong level of evidence, as discussed also in greater detail by Wohlin [128]. Confirmatory research, in turn, typically forms a set of predictions based on a theory (often in the form of propositions or hypotheses) and validates to which extent those predictions hold true or not against empirical observations (thus testing the consequences of a theory). To efficiently build a robust body of knowledge, we need both exploratory and confirmatory research to minimize bias and maximize the validity and reliability of our theories efficiently. Figure 4.1 shows this interdependency between exploratory (theory-generating) and confirmatory (theory-testing) research in empirical research.

In this context, we have set out to address a gap by formalizing and empirically validating the theory of code review as a communication network. To that end, we focus on a phenomenon central to any communication network: its capability to spread information among its participants. We refer to this phenomenon as *information diffusion*. From this perspective, we formulate our theory: code review, as a communication network, enables information to spread both widely and quickly among its participants. Whether this theory holds depends on the actual capabilities of code review systems to facilitate such information diffusion. If a given code review system exhibits little or no diffusion at all, it would challenge the general applicability of the theory to practical terms, suggesting that additional constraints, contextual factors, or boundary conditions must be considered to explain under which conditions and how code review functions effectively as a communication network.

In prior work of this line of research (cf. Figure 4.2), we estimated the capabilities of closed-source code review by measuring how widely and how quickly[1] information can spread in code review systems at Microsoft, Spotify, and Trivago. Although the prior study as *in silico* experiment followed, in principle, confirmatory objectives, it did not explicitly formulate a theory or evaluate propositions and focused exclusively on closed-source code review systems—omitting a central phenomenon in software engineering: open-source software development. Open-source software development follow substantially different mechanics, including organizational structure [67] and developer liability and commitment [11], which may affect the communication within code review. We argue that studying and comparing both open-source and closed-source systems is essential for investigating the theory of code review as a communication network through a more holistic and context-sensitive lens.

The study at hands addresses those limitations and shall close our long-term investigations by answering the following two research questions considering both open-spource and closed-source software systems:

RQ 1 How widely can information spread within code review?

RQ 2 How quickly can information spread within code review?

In alignment with the baseline experiment [39], we address the two research questions in an *in silico* experiment that simulates an artificial information diffusion within the code review within three open-source code review systems (Android, Visual Studio Code, and React) and three closed-source code review systems (Microsoft, Spotify, and Trivago), which we already used for the baseline experiment. The simulated information diffusion within the communication networks identifies all minimal time-respecting paths reflecting information diffusing through the communication network

[1] In this study, we use the terms *widely* and *quickly* instead of *far* and *fast*, as used in our prior study, to improve clarity and readability throughout—at the expense of a minor inconsistency between the two studies.

under best-case assumptions. The participants along those minimal time-respecting paths describe how wide information can spread among code review participants (RQ 1), and the minimal topological and temporal distances between participants describe how quickly information spreads (RQ 2). Together, both measures allow us to empirically test core assumptions of the theory of code review as a communication network.

This work contributes to the understanding of code review as a communication network in software engineering in the following ways:

1. *Validating the formalized theory of code review as a communication network*—We provide a large-scale and sophisticated simulation-based empirical validation that code review systems can support wide and fast information diffusion, reinforcing the theory of code review as a communication network.

2. *Comparative Analysis of Open-Source and Closed-Source Code Review Systems*—Our study aims at critically analyzing and comparing different systems to yield a representative picture. This reveals systematic differences between open-source and closed-source code review environments: open-source systems enable faster diffusion, whereas closed-source systems support broader reach.

3. *Demonstrating the scalability of communication in large code review systems*—We show that information diffusion in code review networks scales effectively, with no observed degradation in diffusion range or speed in larger systems, challenging assumptions that scale inherently limits communication efficiency.

4. *Analysis of scientific and practical implications*—We analyze and highlight the key scientific and practical implications of the theory of code review as a communication network, now grounded in a broader and more robust empirical foundation.

5. *Comprehensive Replication Package*—We release a complete, documented replication package including all datasets, thoroughly tested simulation code, and analysis scripts to support transparency, reproducibility, and future research by the community.

In this manuscript, we borrow the definition of code review from the replicated study: We define code review as the informal and asynchronous discussion around a code change among humans. This means older results from formal code inspections and pair programming as an informal but synchronous discussion around a code change among usually two developers are beyond the scope of our study.

As a replication study, our work naturally shares substantial overlap with the original study we replicate [39], including its definitions and the underlying model of information diffusion in code review [41]. Figure 4.2 illustrates the delineation and interplay between the studies, situating our contribution within the broader line of

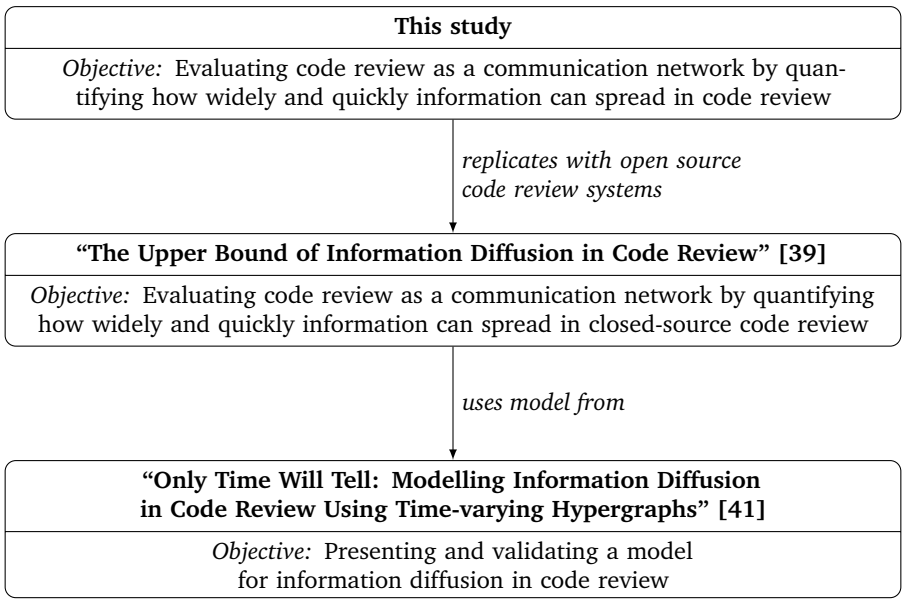


Figure 4.2: The delineation and interplay with our prior work in this line of research.

research. Too often, replication studies in software engineering research still tend to be "not consistent in either the type of information reported or the level of detail reported" [23]. We therefore explicitly decided to provide the full explanations of the simulation model in Section 4.3.2 and to add the inherently overlapping limitations [53] in Section 4.7 to improve the readability and self-containing of this article.

4.2 Background

In the following, we embed this study in the large picture on this line of research (Section 4.2.1), provide background information on replications in software engineering research (Section 4.2.2), position *in silico* experiments as an empirical research method (Section 4.2.3), and discuss prior work on closed-source and open-source code review (Section 4.2.4). For a comprehensive discussion of the related work on measuring information diffusion in code review, we refer the reader to Section 2.2 in Dorner et al. [39].

4.2.1 Line of research

As already highlighted in Figure 4.2, this work forms part of a broader line of research. In the following, we highlight the two preceding studies and their contributions to this line of research.

In the first study of this line of research [41], we presented and evaluated a model for information diffusion in code review. In an *in silico* experiment modelling the code review at Microsoft as our time-varying and as a traditional time-aggregated model, we found that traditional time-aggregated models significantly overestimate the paths of information diffusion available in communication networks and, thus, are neither precise nor accurate for modelling and measuring the spread of information within communication networks that emerge from code review. We will discuss the model as part of Section 4.3.2 in detail.

In our prior work [39], we synthesized the findings from prior qualitative work emerging in the theory of code review as a communication networks. We then used the novel model from [41] to simulate an artificial information diffusion within large (Microsoft), mid-sized (Spotify), and small code review systems (Trivago). We measure the minimal topological and temporal distances between the participants to quantify how widely and how quickly information can spread in code review, which forms the upper bound of information diffusion in code review. This *in silico* experiment serves as the baseline experiment for our replication. As such, we discuss the details of the findings in detail as part of our Section 4.5. Despite its confirmatory nature and the research design as an experiment, we lost out on explicitly framing this study as a confirmatory study with explicit propositions. We remedy this shortcoming in this study.

4.2.2 Replications in Software Engineering

To consolidate a body of knowledge built upon evidence, experimental results have to be extensively verified. Replications can serve this purpose and verify the validity and reliability of previous findings.

Although replications are a cornerstone in science and replication studies in software engineering research are generally gaining much attention in recent years [28], there is no common interdisciplinary (let alone intradisciplinary) understanding of replications [54][2].

In our work, we follow the definitions and classification of Gómez, Juristo, and Vegas. In their, to us inspiring work, the authors proposed a classification of replication types within software engineering research [53]. They identified three different types of replications: literal, operational, and conceptual replications. Operational replications can be altered by four dimensions of experimental configurations: protocol, operationalizations, populations, and experimenters.

Following this classification, our study is an operational replication where we varied the population to verify the limits of the population of code review systems used

[2]We deem to mention at this point that ACM swapped the terms “reproducibility” and “replication” in August 2020 to “harmonize its terminology and definitions with those used in the broader scientific research community” [27].

Table 4.1: Replication types for experiments in software engineering (adapted from [53]). This study is an operational replication in which we varied the population (highlighted in **bold**).

Type	Varied dimension	Description
Literal	None	Repetition. The aim is to run a replication of the baseline experiment as exactly as possible. The replication is run by the same experimenters using the same protocol and the same operationalizations on different samples of the same population.
Operational	Protocol	The experimental protocol elements are varied with the aim of verifying that the observed results are reproduced using equivalent experimental protocols.
	Operationalization	The cause and/or effect operationalizations are varied in order to verify the bounds of the cause and/or effect construct operationalizations within which the results hold.
	Population	**The populations are varied to verify the limits of the populations used in the baseline experiment.**
	Experimenter	The experimenters are varied to verify their influence on the results.
Conceptual	Unknown	Reproduction. Different experimenters run the replication with new protocols and operationalizations.

in the baseline experiment. Gómez, Juristo, and Vegas suggests the name *changed-populations replication* (highlighted in Table 4.1) for our type of replication.

As already pointed out in Section 4.1, replication studies in software engineering still tend to be "not consistent in either the type of information reported or the level of detail reported" [23]. Carver proposed a starting point for a discussion on reporting guidelines for experimental replications in software engineering research [23]. For our study, we follow those reporting guidelines as we include the key aspects of the original study, report the motivation for conducting the replication (Section 4.1), discuss the threats of validity of the overlapping in experimenters in both experiments (Section 4.7), and we will compare the results of the original study as part of the discussion (Section 4.5) which serves as foundation for validating our theory.

4.2.3 In silico Experiments in Software Engineering

In this *in silico* experiment, we simulate an artificial information diffusion and measure the resulting traces generated by the spread of the information [39]. Given the rarity of simulations in the empirical software engineering community, we motivate the explicit choice of that as our empirical research method.

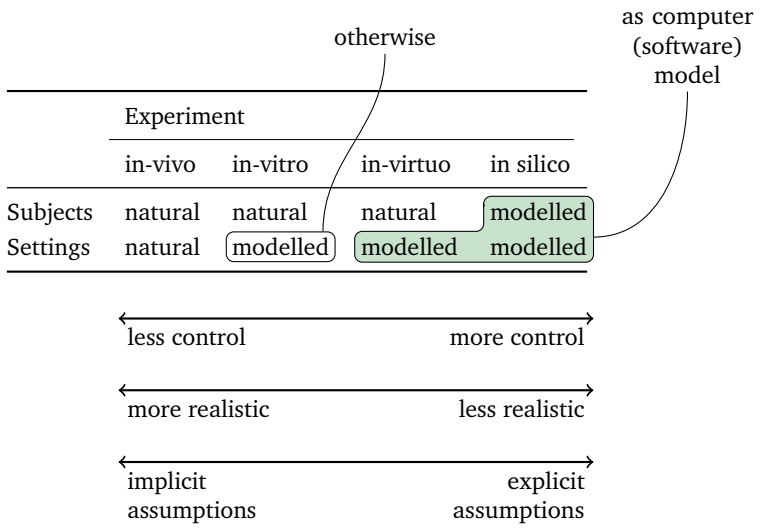
Simulations are experiments with a model. Simulations play to their strengths when traditional experiments would have been too complex, too expensive, too lengthy, or simply not possible or accessible otherwise. Those attributes match the objectives and setting of our research since experiments with whole software companies or open-source components are not feasible.

If all parts of the experiment, i.e. subjects and settings, are modelled as software (see Table 4.2), the simulation becomes an *in silico* experiment.

Although simulations have been applied in different research fields of software engineering, e.g., process engineering, risk management, and quality assurance [81], the role of simulations as an empirical method is still often subject to some form of prejudice but also subject to ongoing more philosophical debates. Stol and Fitzgerald, for example, positioned computer simulations in their ABC framework [114] in a non-empirical setting because, as the authors argue: "while variables can be modelled and manipulated based on the rules that are defined within the computer simulation, the researcher does not make any new empirical observations of the behavior of outside actors in a real-world setting (whether these are human participants or systems)" [114]. Without discussing the role of simulations in the empirical software engineering community to the extent they might deserve, however, we still argue for their suitability as an evidence-based (empirical) approach in our context where observations would otherwise not be possible (or, at least, not realistic).

We consider computer simulations as an empirical research method, the same as done in other disciplines and inter-disciplines (where, for instance, climate simulations are

Table 4.2: A comparison of *in-vivo*, *in-vitro*, *in-virtuo*, and *in silico* experiments with respect to subjects and settings.



the first-class citizens in the set of research methods). Empirical research methods are "research approaches to gather observations and evidence from the real world" [114] and same as in other empirical research methods, in simulation models, we build the models based on real-world observations and make conclusions based on the empirical observations along the execution (in our case, of the simulations). These simulation models are abstractions from the real world—same as the (often implicit) theoretical models underlying quasi-controlled (in-vitro) experiments. Simulations and their underlying models further abstract from (and make explicit) complex systems and make observations and evidence possible in situations where more traditional experiments are rendered infeasible (e.g., too expensive, dangerous, too long, or not accessible) or simply impossible at all; for instance, observations when exploring the capabilities of real-world communication networks with thousands of developers as done in the simulation study presented in the manuscript at hands.

Needless to elaborate, a certain abstraction from the real world is inherent to all empirical research methods, either in the form of explicit models or implicit assumptions. Like every measurement, the models we create come with certain accuracy and precision—with a certain quality. However, we may still argue that the quality of a research method does not necessarily decide upon whether it qualifies as empiricism or not but rather the underlying constructs and their (evidence-based) sources. To avoid surreal models and ensure the quality of a model, however, the modelling itself needs to be guided by quality assurance in verification and validation, and the sample used needs to be realistic; both would, in turn, be in tune with the underlying arguments by available positionings such as the one by [114]. To increase the transparency in

the quality of our simulations, we further disclose all developed software components as a replication package, also including the open-source communication networks we used as a sample.

4.2.4 Open-Source vs. Closed-Source Code Review

Insights and results from open source are not easily transferrable to closed-source software development in general since the definition does not refer to a specific software development process. Open-source software is software that is available under a license that grants the right to use, modify, and distribute the software—modified or not—to everyone free of charge [adapted from 97]. This definition does not imply any specific software development or quality assurance process. However, the public availability of the source code alone is a necessary but still not sufficient condition.

The right to modify open-source code makes code review a cornerstone for quality assurance in open source: Open-source projects try to feed back modifications to benefit from improving or enhancing their projects and, thereby, create a larger community around the open-source project. However, unlike closed-source software development, where developers have an employment agreement, a developer contributing to an open-source project has no formal contract with the open-source project specifying conditions, obligations, and commitment. Contributors in open source may make contributions only once, for a short duration, or infrequently and irregularly [11, 12, 70]. Open-source project maintainers may have to maintain the source code contributed by themselves or find a developer who can, which can be difficult without formal managerial authority over open-source developers. Therefore, the lack of formal managerial authority leaves code review the only way for open-source maintainers to exercise power over open-source developers. This underlying motivation makes any comparison of open-source and closed-source code review inherently challenging.

Two seminal studies accepted the challenge and compared code review in open-source and closed-source software development with respect to code review practices and quantitative measurements on the one hand and expectations towards code review on the other.

First, Rigby and Bird compared code review practices from open-source and closed-source software development efforts in order to distill code review practices [99]. The sample of code review systems ranges from closed-source software development projects at Microsoft (Bing, SQL Server, and Office 2013) and AMD, as well as open-source code review systems. The authors implicitly distinguish between community-led (i.e., Apache, Subversion, Linux, FreeBSD, KDE, and Gnome) and Google-led

open-source projects (Android[3] and Chromium OS[4]). Although we explicitly exclude code review practices as a sampling dimension for our approach, narrowing down the large diversity within open source to company-led open-source projects allows us to focus on relevant and reasonably large open-source projects. Surprisingly, the study finds little difference between the different code review systems studied in the context of open-source or closed-source software development.

Second, Bosu et al. reports the findings of an interview study with a well-designed survey of code review participants from open-source and closed-source software development, namely Microsoft, [17]. They mined the code review systems of 34[5] open-source projects "that used either Gerrit, ReviewBoard, or Rietveld to identify who had participated in at least 30 code review requests". Unfortunately, we could not retrieve how the sampling frame [9] of open-source projects was selected or constructed.

Both studies found a large overlap between code review in the context of open-source and closed-source software development with respect to their respective research area. Rigby and Bird finds little difference between the different code review systems studied in the context of open-source or closed-source software development with respect to different measurements of code review systems in both worlds [99]. [17] also finds that "the results were similar for the [open-source] and Microsoft developers." with the following two notable exceptions: The participants from open source reported building a relationship between the code review participants is the most important aspect while code review participants from Microsoft consider information diffusion ("knowledge dissemination") more important. Similarly, open-source code review participants take the relationship with the code author and his or her reputation into account most prominently to decide whether to contribute to a code review. Conversely, for code review participants at Microsoft, the expertise of the code author (i.e., if a developer writes good code that s/he can learn) is the most important aspect.

In our manuscript, we will continue and extend this discussion on the equivalence of code review in closed-source and open-source software development to information diffusion within code review.

4.3 Experimental Design

In this section, we describe the design of our *in silico* experiment that evaluates and quantifies how widely (RQ 1) and how quickly (RQ 2) information can diffuse in code review and postulate our propositions.

[3]Throughout this paper, we refer to the Android Open Source Project simply as *Android*.

[4]The open-source project that backs the operating system *Chrome OS* is officially named *Chromium OS*. We took the liberty to use the official name here.

[5]The introduction speaks of 36 open-source projects.

The underlying idea of our *in silico* experiment is, in principle, simple: We model different code review systems as a communication networks and compute all minimal distances between all code review participants. The cardinality of reachable participants indicates how wide (RQ 1) information can spread, and distances between participants indicate how quickly (RQ 2) information can spread in code review. Since we used minimal paths and created the communication networks under best-case assumptions, the results describe the upper bound of information diffusion in code review [39].

Yet communication—and, by extension, information diffusion—is (1) inherently time-dependent and (2) not strictly bilateral, as code reviews often involve exchanges among more than two participants. As a result, traditional graphs are not well suited for modelling such interactions and tend to dramatically overestimate information diffusion [41]. To address this, we use time-varying hypergraphs to model the communication network and to compute the shortest paths between all vertices. Hypergraphs generalize traditional graphs, allowing the use of standard algorithms such as Dijkstra's for computing minimal-path distances. However, in time-varying hypergraphs, the distance between two vertices can be both topological (i.e., the fewest hops) and temporal (i.e., the shortest duration). This dual characterization enables us to answer RQ 2 by measuring both types of distance. Importantly, both topological and temporal distances yield the same set of reachable participants, which serves as the basis for answering RQ 1.

Since all models are per definition abstractions and, accordingly, simplifications of reality, the quality of an *in silico* experiment highly depends on the quality of the simulation model and its parametrization. Therefore, we provide a more elaborate description of our simulation model, which was originally proposed and partially validated in [41], in Section 4.3.2 and its parametrization of its computer model by empirical code review data (Section 4.3.3). Before that, however, we first derive and explicitly state our propositions the theory consists of in greater detail.

4.3.1 Propositions

If code review is a functional communication network that enables the exchange of information (theory T) as identified by different exploratory studies [39], then code review spreads information quickly (proposition P_2) and widely (proposition P_1) among its participants. We can formulate this sentence as the propositional statement

$$T \implies (P_1 \wedge P_2)\,. \tag{4.1}$$

That means our theory T can be falsified in its universality if at least one of our empirically testable propositions derived from the theory of code review as a communication network is contradicted by the data. Note that our goal is not primarily to reach

a binary conclusion by falsifying a theory through classical, statistical hypothesis testing, but rather to gain deeper insights through an experimental and confirmatory approach. Since there is no predefined threshold for rejecting our propositions, we adopt a data-driven interpretation grounded in observed patterns. Rather than setting an arbitrary cutoff *a priori*, we analyze the empirical upper bound of information diffusion within our sample of code review systems to identify consistent deviations from the theoretical expectations encoded in the propositions.

4.3.2 Simulation Model

In general, a simulation model consists of two components: (1) the conceptual model describing our derivations and assumptions and (2) the computer model as the implementation of the conceptual model. The following two subsections describe each component in detail.

Conceptual Model

In the following, we describe how we conceptually model communication networks from code review discussions and the information diffusion within those communication networks.

Communication Network Communication, the purposeful, intentional, and active exchange of information among humans, does not happen in the void. It requires a channel to exchange information. A *communication channel* is a conduit for exchanging information among communication participants. Those channels are

1. *multiplexing*—A channel connects all communication participants sending and receiving information.
2. *reciprocal*—The sender of information also receives information and the receiver also sends information. The information exchange converges. This can be in the form of feedback, queries, or acknowledgments. Pure broadcasting without any form of feedback does not satisfy our definition of communication.
3. *concurrent*—Although a human can only feed into and consume from one channel at a time, multiple concurrent channels are usually used.
4. *time-dependent*—Channels are not available all the time; after the information is transmitted, the channels are closed.

Channels group and structure the information for the communication participants over time and content. Over time, the set of all communication channels forms a communication network among the communication participants.

In the context of our study on information diffusion, a communication channel is a discussion in a merge (or pull) request. A channel for a code review on a merge

request begins with the initial submission and ends with the merge in case of an acceptance or a rejection. All participants of the review of the merge request feed information into the channel and, thereby, are connected through this channel and exchange information they communicate. After the code review is completed and the discussion has converged, the channel is closed and archived, and no new information becomes explicit and could emerge. However, a closed channel is usually not deleted but archived and is still available for passive information gathering. We do not intend to model this passive absorption of information from archived channels by retrospection with our model. For this line of research, we recommend the work by [92] as further reading.

From the previous postulates on channel-based communication, we derive our mathematical model: Each communication medium forms an undirected, time-varying hypergraph in which hyperedges represent communication channels. Those hyperedges are available over time and make the hypergraph time-dependent. Additionally, we allow parallel hyperedges[6]—although unlikely, multiple parallel communication channels can emerge between the same participants at the same time but in different contexts.

Such an undirected, time-varying hypergraph reflects all four basic attributes of channel-based communication:

- *multiplexing*—since a single hyperedge connects multiple vertices,
- *concurrent*—since (multi-)hypergraphs allow parallel hyperedges,
- *reciprocal*—since the hypergraph is undirected, information is exchanged in both directions, and
- *time-dependent*—since the hypergraph is time-varying.

In detail, we define the channel-based communication model for information diffusion in the timeframe $\mathcal{T}$ to be an undirected time-varying hypergraph

$$\mathcal{H} = (V, \mathcal{E}, \rho, \xi, \psi)$$

where

- V is the set of all human participants in the communication as vertices
- $\mathcal{E}$ is a multiset (parallel edges are permitted) of all communication channels as hyperedges,

[6]This makes the hypergraph formally a *multi-hypergraph* [90]. However, we consider the difference between a hypergraph and a multi-hypergraph as marginal since it is grounded in set theory. Sets do not allow multiple instances of the elements. Therefore, instead of a set of hyperedges, we use a multiset of hyperedges that allows multiple instances of the hyperedge.

- ρ is the *hyperedge presence function* indicating whether a communication channel is active at a given time,
- $\xi : E \times \mathcal{T} \to \mathbb{T}$, called *latency function*, indicating the duration to exchange information among communication participants within a communication channel (hyperedge),
- $\psi : V \times \mathcal{T} \to \{0, 1\}$, called *vertex presence function*, indicating whether a given vertex is available at a given time.

Information Diffusion The time-respecting routes through the communication network are potential *information diffusion*, the spread of information. To estimate the upper bound of information diffusion and, thereby, answer both of our research questions, we measure the distances between the participants under best-case assumptions.

For information diffusion in code review, we made the following assumptions:

- *Channel-based*—Information can only be exchanged along the information channels that emerged from code review. The information exchange is considered to be completed when the channel is closed.
- *Perfect caching*—All code review participants can remember and cache all information in all code reviews they participate in within the considered time frame.
- *Perfect diffusion*—All participants instantly pass on information at any occasion in all available communication channels in code review.
- *Information diffusion only in code review*—For this simulation, we assume that information gained from discussions in code review diffuses only through code review.
- *Information availability*—To have a common starting point and make the results comparable, the information to be diffused in the network is already available to the participant, which is the origin of the information diffusion process.

Our assumptions make the results of the information diffusion a best-case scenario. Although the assumptions do not likely result in actual, real-world information diffusion, they serve well the scope of our study, namely to quantify the upper bound of information diffusion.

The possible routes through the communication network describe how information can spread through a communication network. Those routes are time-sensitive: a piece of information gained from a communication channel (i.e., a code review discussion) can be shared and exchanged in all subsequent communication channels but not in prior, closed communication channels.

Mathematically, those routes are time-respecting walks, so-called *journeys*, in a time-varying hypergraph representing the communication network. A journey is a sequence of tuples

$$\mathcal{J} = \{(e_1, t_1), (e_2, t_2), \ldots, (e_k, t_k), \}$$

such that $\{e_1, e_2, \ldots, e_k\}$ is a walk in $\mathcal{H}$ with $\rho(e_i, t_i) = 1$ and $t_{i+1} > t_i + \xi(e_i, t_i)$ for all $i < k$.

We define $\mathcal{J}^*_{\mathcal{H}}$ the set of all possible journeys in a time-varying graph $\mathcal{H}$ and $\mathcal{J}^*_{(u,v)} \in \mathcal{J}^*_{\mathcal{H}}$ the journeys between vertices u and v. If $\mathcal{J}^*_{(u,v)} \neq \emptyset$, u can reach v, or in short notation $u \rightsquigarrow v$.[7] Given a vertex u, the set $\{v \in V : u \rightsquigarrow v\}$ is called *horizon* of vertex u.

The notion of length of a journey in time-varying hypergraphs is two-fold: Each journey has a topological distance (measured in number of hops) and temporal distance (measured in time). This gives rise to two distinct definitions of distance in a time-varying graph $\mathcal{H}$:

- The *topological distance* from a vertex u to a vertex v at time t is defined by $d_{u,t}(v) = \min\{|\mathcal{J}(u, v)|_h\}$ where the journey length is $|\mathcal{J}(u, v)|_h = |\{e_1, e_2, \ldots, e_k\}|$. This journey is the *shortest*.
- The *temporal distance* from a vertex u to a vertex v at time t is defined by $\hat{d}_{u,t}(v) = \min\{\psi(e_k) + \xi(e_k) - \xi(e_1)\}$.[8] This journey is the *fastest*.[9]

With this conceptual model and its mathematical background, we are now able to answer both research questions by measuring two characteristics of all possible routes through the communication network:

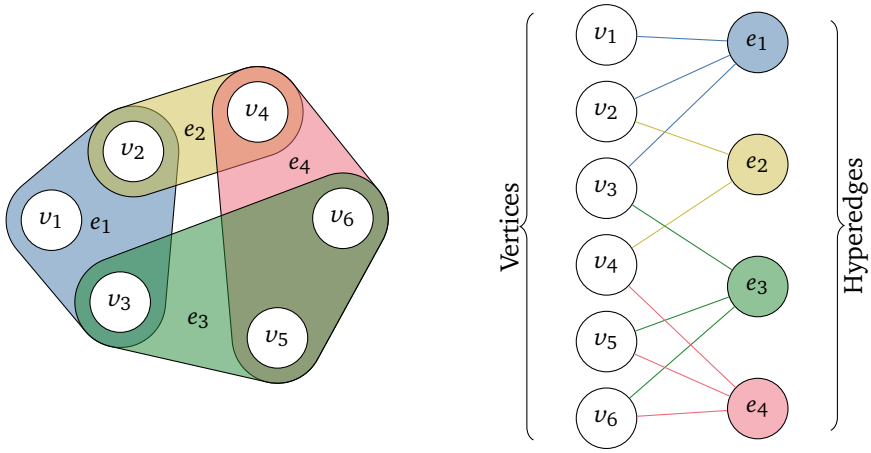
- The distribution of the horizon of each participant in a communication network represents how wide information can spread (RQ 1).
- The distribution of all shortest and fastest journeys between all participants in a communication network answers how fast information can spread in code review (RQ 2). We measure how fast information can spread in code review in terms of the topological distance (minimal number of code reviews required to spread information between two code review participants) and the temporal distance (minimal timespan to spread information between two code review participants).

Those measurements within code review communication networks will result in the upper bound of information diffusion in code review.

[7] In general, journeys are not symmetric and transitive—regardless of whether the hypergraph is directed or undirected: $u \rightsquigarrow v \not\Leftrightarrow v \rightsquigarrow u$.

[8] In our case, $\psi(e_k)$ is always 0.

[9] For the interested reader, we would like to add that if the temporal distance is not defined for a relative time but for an absolute time $\hat{d}_{u,t}(v) = \min\{\psi(e_k) + \xi(e_k)\}$, the journey is called *foremost*. For this line of research, the foremost journeys are not used.



(a) An example time-varying hypergraph whose so-called hyperedges (denoted by $e_\square$) can link any arbitrary number of vertices (denoted by $v_\square$): For example, hyperedge e_3 connects three vertices. The horizon and minimal paths of vertex depend highly on the temporal order of the hyperedges: for example, the horizon of v_1 contains all vertices if the temporal availabilities of the hyperedges are $e_1 < e_2 < e_4 < e_3$, but none if $e_1 > e_2 \geq e_3$.

(b) Any hypergraph can be transformed into an equivalent bipartite graph: The hyperedges and the vertices from the time-varying hypergraph from (a) become the two distinct sets of vertices of a bipartite graph.

Figure 4.3: An example hypergraph (a) and its bipartite-graph equivalent (b).

Computer Model

Since our mathematical model is not trivial and lacks performant tool support for time-varying hypergraphs, we dedicate this section to the computer model and the implementation of the mathematical model described previously.

Time-varying hypergraphs are a novel concept; therefore, we cannot rely on existing toolings. We implemented the time-hypergraph as an equivalent bipartite graph: The hypergraph vertices and hyperedges become two sets of vertices of the bipartite graph. The vertices of those disjoint sets are connected if a hypergraph edge is part of the hyperedge. Figure 4.3 shows a graphical description of the equivalence of hypergraphs and bipartite graphs.

Since the quality of the experiment and its outcome heavily relies on *in silico* experiments, we developed and validated a suitable simulation model for conducting such *in silico* experiments [41].[10]

[10]As part of our simulation model, we established the theoretical foundation for a novel generalization of Dijkstra's algorithm for shortest paths in time-varying hypergraphs and its first implemention. To validate

We use a modified Dijkstra's algorithm to find the minimal journeys for each vertex (participant) in the time-varying hypergraph. Dijkstra's algorithm is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. In contrast to its original form, our implementation finds both the shortest (a topological distance) and fastest (a temporal distance) journeys in time-varying hypergraphs.[11] Since Dijkstra's algorithm can be seen as a generalization of a breadth-first search for unweighted graphs, we can identify not only the minimal paths but also the horizon of each participant in the communication network in one computation.

The algorithm is integrated into our computer model and implemented in Python. For more implementation details and performance considerations, we refer the reader to our replication package, including its documentation [33, 34]. Since both time-varying hypergraphs as a data model and the generalized Dijkstra's algorithm for time-varying hypergraphs are novel contributions, we developed an extensive test suite to ensure that the computational model accurately reflects the underlying mathematical framework and that our Dijkstra implementation produces correct results. This effort was carried out in close collaboration with students from a software testing course at BTH. We documented our experiences in a dedicated study [35]. Additionally, the model parameterization and simulation code [34] as well as the intermediate and final results [33] are publicly available under an open-source license.

4.3.3 Model Parametrization

Instead of a theoretical or probabilistic parametrization, we parametrize our simulation model with empirical code review systems from three open-code review systems: Android, Visual Studio Code, and React.

In the following, we describe our sampling strategy for selecting suitable open-source code review systems and the code review data extraction process for parametrizing our simulation model.

Sampling

We use a *maximum variation sampling* to select suitable code review communication networks in open source. A maximum (or maximum heterogeneity) variation sampling is a non-probabilistic, purposive sampling technique that chooses a sample to maximize the range of perspectives investigated in the study in order to identify important shared patterns that cut across cases and derive their significance from having emerged out of heterogeneity [117].

our Python implementation, we developed a extensive test suite in close collaboration with students from a software testing course at Blekinge Institute of Technology, Sweden. We reported our experiences in dedicated study [35].

[11] For future applications, our implementation of Dijkstra's algorithm can also find any foremost journey.

To ensure comparability with the replicated study, we sampled for the following two perspectives:

- *Code review system size*—To avoid a bias introduced by network effects, we required communication networks emerging from different sizes of code review. The size of a communication network can be measured in terms of the number (hyper)edges (corresponding to the number of code reviews) or vertices (corresponding to the number of participants). In our sample, we use a small (React), mid-sized (Visual Studio Code), and large (Android) code review system (see Table 4.3). The size classification in small, mid-sized, and large code review systems is arguably arbitrary and relative to the code review systems in our sample rather than following a general norm that, to the best of our knowledge, does not exist.
- *Code review tool*—In particular, since Baum et al. suggested code review tool in use as a main factor shaping code review in industry [14], we aim to minimize the code review tool bias for the results and require our sample to contain a diverse set of code review tools. Our sample contains three different code review tools: BitBucket, GitHub, and CodeFlow.

In alignment with the qualitative prior work, we explicitly excluded the different manifestations in code review practices as a sampling dimension.

We restrict the population of open-source projects to active and industrial-relevant open-source projects, which are, in our definition, initiated and predominantly led by companies under ongoing development as of this study (at the time of writing this manuscript in 2025). This restriction originates mainly in the heterogeneity of open source: The definition of open-source software refers only to its licensing and does not imply a particular software development method. Thus, there is no single, unified open-source software development and quality assurance process, which makes a comparison of open-source projects inherently difficult. By restricting our population to company-led open-source projects, we aim to reduce this heterogeneity in open-source and make our sample more homogenous.

From this population, we drew a sample of three open-source projects: Android, Visual Studio Code, and React. Those projects are all driven by large software companies, i.e., Google, Microsoft, and Facebook.[12] Table 4.3 provides an overview of our sample of code review systems and the dimension of representativeness. We describe the cases in our sample in more detail in the following subsections.

Android Open Source Project The Android Open Source Project (AOSP) is an initiative led by Google to develop and maintain open-source software for mobile

[12]Our replication package includes all necessary software to collect and analyze data from code review systems beyond our sample, ensuring straightforward and efficient replication outside the scope of this study.

Table 4.3: Our sample of open-source code review systems with respect to the two dimensions of representativeness during the timeframe under investigation: code review system size and tooling. For completeness and ease of reference, we included the sample of closed-source code review systems from the replicated study [39].

Code review system	Size	Code reviews	Participants	Tooling
Open source				
Android	large	10 279	1793	Gerrit
Visual Studio Code	mid-sized	802	162	GitHub
React	small	229	64	GitHub
Closed source (via replicated study)				
Microsoft	large	309 740	37 103	CodeFlow
Spotify	mid-sized	22 504	1730	GitHub
Trivago	small	2442	364	BitBucket

devices. It provides the source code for building Android firmware and apps, including the operating system framework, Linux kernel, middleware, and essential system applications. The AOSP serves two purposes: First, as an open-source project, the AOSP offers a platform that is open and accessible to developers, allowing them to customize and extend Android for various devices and use cases. This openness has fostered a vibrant ecosystem of custom ROMs, modifications, and alternative distributions of Android. Second, the AOSP is the reference implementation of *Android*, providing a standard for device manufacturers and developers to follow when creating Android-based products and applications. While Google's own Android releases (such as those on Pixel devices) often include proprietary Google services and applications, AOSP itself is fully open source and can be used as the basis for building Android variants without Google's involvement. The project uses *Gerrit*[13] as code review tool, which mirrored Google's internal code review tool at the time *Mondrian*. The code review within Android Open Source Project is intensively studied [5, 15, 43, 44, 57, 80, 89, 92, 99, 119, 120, 121, 132, 135] which makes it probably the well-known code review system in software engineering research. Throughout this paper, we will refer to the Android Open Source Project simply as *Android*.

Visual Studio Code Visual Studio Code, often abbreviated as VS Code, is a free, open-source code editor initiated and orginally developed by Microsoft. It's widely used by developers for various programming languages and platforms. It's known for its lightweight and fast performance, making it a popular choice among developers for writing code, debugging, and managing projects across different platforms. Visual Studio Code uses GitHub through its pull-request functionality as code review tool.

[13] `https://www.gerritcodereview.com`

Table 4.4: Timeframes and the data collection timeframe among our cases.

	Timeframe (four weeks)	Collected during
Android	2024-03-04 to 2024-03-31	April 2025
Visual Studio Code	2024-03-04 to 2024-03-31	April 2025
React	2024-03-04 to 2024-03-31	April 2025

React React is a popular JavaScript library for building user interfaces, particularly for web applications. The open-source project was initiated and is led by Facebook. React is known for its declarative and component-based approach to building UIs. It allows developers to create reusable UI components that manage their own state, making it easier to build complex UIs with minimal code duplication. Like Visual Studio Code, React uses GitHub through its pull-request functionality as code review tool.

Data Collection

We extract all human interactions with the code review discussions within four consecutive calendar weeks from the single, central code review tools in each open-source software development context.

We define a code review interaction as any non-trivial contribution to the code review discussion: creating, editing, approving or closing, and commenting on a code review. For this study, we do not consider other (tool-specific) types of discussion contributions (for example, emojis or likes) a substantial contribution to a code review.

The beginning and end of those four-week timeframes differ and are arbitrary, but share the common attributes: All timeframes

- start on a Monday and end on a Sunday,
- have no significant discontinuities by public holidays such as Christmas or Midsommar,
- are pre-pandemic to avoid introduced noise from changes in remote work policies, pandemic-related restrictions, or interferences in the software development.

Table 4.4 lists the timeframes (each four weeks) and when the data was collected.

We excluded all non-human code-review participants and interactions (i.e., bots or automated tasks contributing to the code-review discussions). To protect the privacy of all individuals, we strictly anonymized all participants and removed all identifiable personal information.

We made all data and results, as well as the extraction pipeline for Gerrit and GitHub, publicly available.

4.4 Results

This section presents four measurements from our simulation, described in Section 4.3, structured around our two research questions: how widely (RQ 1) and how quickly (RQ 2) information can spread in code review under best-case assumptions.

To address RQ 1, we present the measurements on the absolute (Section 4.4.1) and normalized information diffusion range (Section 4.4.1), capturing how far information can reach across participants. For RQ 2, we examine the topological (Section 4.4.2) and temporal distances between participants (Section 4.4.2), reflecting the speed of potential information diffusion in terms of structure and time.

To enable clearer and more direct comparisons, we present the results from the open-source code review systems alongside findings from a replicated study that examines the upper bound of information diffusion in closed-source code review systems at large (Microsoft), mid-sized (Spotify), and small (Trivago) companies. The colors representing the companies is consistent throughout this paper. Values in figures and tables are normalized to the $[0, 1]$ range, while percentages are used in the text for readability.

4.4.1 How wide can information spread within code review (RQ 1)?

As outlined in Section 4.3.2, we address RQ 1 by measuring how many participants each individual can reach in the code review communication network. Mathematically, this is the cardinality of a participant's horizon, which we refer to as the *information diffusion range*. We report both the normalized and absolute diffusion range.

Normalized Information Diffusion Range

To make the different code review system sizes comparable, we normalize the information diffusion range to the number of code review participants in a code review system. Mathematically, we define the normalized information diffusion range for all code review participants $u \in V$ by

$$\frac{|\{v \in V : u \rightsquigarrow v\}|}{|V|}.$$

Figure 4.4 plots the empirical cumulative distribution functions (ECDF) visualizing the distributions of the normalized information diffusion range per participant after four weeks each resulting from our simulation.

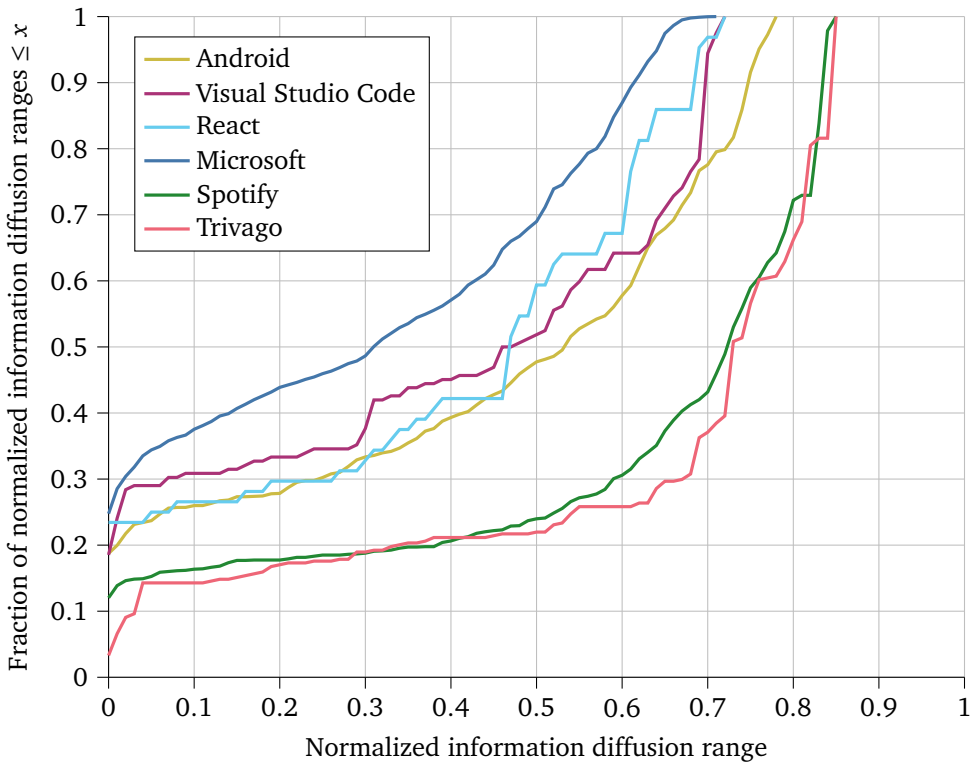


Figure 4.4: Empirical cumulative distribution of the normalized information diffusion ranges per code review system after four weeks.

Table 4.5: Quantile ranges of the normalized information diffusion ranges per participant for each code review system.

	Quantile range			
	0.3 to 1.0	0.5 to 1.0	0.7 to 1.0	0.9 to 1.0
Android	0.23 to 0.78	0.53 to 0.78	0.66 to 0.78	0.74 to 0.78
Visual Studio Code	0.03 to 0.72	0.45 to 0.72	0.64 to 0.72	0.69 to 0.72
React	0.19 to 0.72	0.39 to 0.72	0.58 to 0.72	0.64 to 0.72
Microsoft	0.01 to 0.71	0.30 to 0.71	0.50 to 0.71	0.61 to 0.71
Spotify	0.58 to 0.85	0.72 to 0.85	0.79 to 0.85	0.83 to 0.85
Trivago	0.67 to 0.85	0.72 to 0.85	0.81 to 0.85	0.83 to 0.85

We identified the upper bound of normalized information diffusion range within closed-source code review systems at both Trivago and Spotify. Interestingly, the two platforms exhibit nearly identical distributions in terms of how widely participants can spread information relative to the size of their respective networks. Specifically, at Trivago, a code review participant can reach up to 85 % of their network. Furthermore, 30 % of participants are able to reach between 81 % and 85 % of all participants, while the median participant can reach between 72 % and 85 % of all participants within the network.

No open-source code review system exceeded the upper bound found in the closed-source code review systems—lending strong support to the findings of the replicated study. We observed the highest normalized information diffusion range in open-source systems at Android: A code review participants in the code review systems of Android can reach 78 % of its network at maximum. For a detailed comparison, we refer to Table 4.5, which presents the upper quantile ranges of information diffusion—specifically, the 0.3 to 1.0, 0.5 to 1.0, 0.7 to 1.0, and 0.9 to 1.0 ranges—in each code review system.

Although Figure 4.4 reveals a comparable pattern in the normalized information diffusion range—reflecting how widely information can spread during code review—in the three open-source code review systems, we could not find a clear explanation for them. Neither the network size (measured by the number of code reviews or participants) nor the specific code review tool accounts for these patterns. For example, despite the significant difference in size (37 104 code review participants at Microsoft and 162 code review participants at Visual Studio Code), the normalized information diffusion range in Microsoft's closed-source code review systems and the open-source system for Visual Studio Code shows a comparable distribution of reachable participants relative to system size. With a median of 45 % and 72 % at

Visual Studio Code and 30 % and 71 % of reachable participants, both code review systems define the smallest diffusion range in our sample.

Absolute Information Diffusion Range

If we consider the absolute information diffusion range for each code instead, which is defined for all code review participants $u \in V$ by

$$|\{v \in V : u \rightsquigarrow v\}|,$$

code review participants at Microsoft's code review system can reach the most participants. Although Microsoft's code review system, along with Visual Studio Code's, exhibits the smallest normalized information diffusion range, it sets the upper bound for the absolute information diffusion range in our sample. In detail, the code review system at Microsoft can spread information up to 26 216 participants (71 % of the total network size), half of the code review participants can reach 11 645 or more other participants.Table 4.6 lists the ranges of the absolute information diffusion range we found for the quantile ranges 0.7, 0.5, 0.3 and 0.1

Among open source projects, the upper bound of the absolute information diffusion range is substantially smaller. For example, half of the participants in Android's code review system are exposed to information from between 966 and 1407 other participants. Even considering the exceptionally large size at Microsoft, Android and Spotify have a comparable number of code review participants (1793 at Android and 1730 at Spotify), the absolute information diffusion range is notably more constrained in Android.

4.4.2 How quickly can information spread within code review (RQ 2)?

As outlined in Section 4.3.2, we address RQ 2 by analyzing the distances between code review participants. In time-varying hypergraphs, distance is characterized in two distinct ways: topological distance, defined as the minimal number of hops across all time-respecting paths (or journeys), and temporal distance, defined as the minimal duration of those time-respecting paths (or journeys).

Accordingly, our answers to RQ 2 are structured around these two complementary notions of distance.

Topological Distances in Code Review

Figure 4.5 illustrates the empirical cumulative distribution of topological distances among code review participants in the analyzed code review systems.

In comparison to our prior findings, the open-source code review systems of React and Visual Studio Code extend the upper bound of how quickly information can

Table 4.6: Quantile ranges of the absolute information diffusion ranges per participant for each code review system.

	Quantile range			
	0.3 to 1.0	0.5 to 1.0	0.7 to 1.0	0.9 to 1.0
Android	448 to 1407	966 to 1407	1197 to 1407	1344 to 1407
Visual Studio Code	5 to 116	74 to 116	104 to 116	113 to 116
React	12 to 46	25 to 46	37 to 46	41 to 46
Microsoft	808 to 26 216	11 645 to 26 216	18 887 to 26 216	22 983 to 26 216
Spotify	1026 to 1472	1260 to 1472	1386 to 1472	1447 to 1472
Trivago	245 to 310	266 to 310	296 to 310	309 to 310

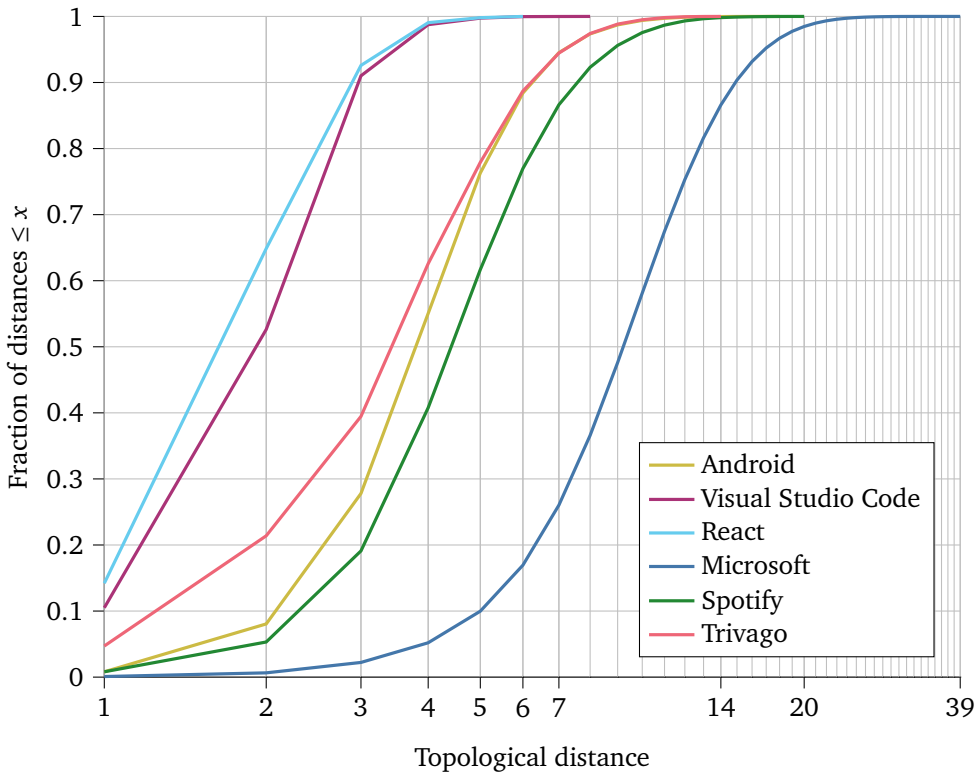


Figure 4.5: Empirical cumulative distribution of the topological distances between participants per code review system after four weeks. The topological distance is the minimal number of code reviews (hops) required to spread information from one code review participant to another.

spread through code review. Specifically, the average (median) distance between two arbitrary participants is one for React and Visual Studio Code, for Android it is three. In contrast, the average (median) distances at Trivago, Spotify, and Microsoft are three, four, and nine[14] hops, respectively. The maximum observed distances are 14 hops for Trivago, 20 hops for Spotify, and 39 hops for Microsoft[14]. Open source code review systems again are significant shorter and have a maximum observed distance of 18 (for Android), 8 (for Visual Studio Code), and 6 (for React).

Temporal Distances in Code Review

The other type of distance in time-varying hypergraphs is temporal distance. The fastest time-varying path refers to the path between two code review participants that minimizes the (relative) temporal distance; i.e., the shortest timespan required for information to spread from one participant to another. Given our observation window, the temporal distance in our measurements cannot exceed four weeks. Figure 4.6 shows the cumulative distribution of the relative temporal distances between the code review participants in our sample.

Again, an open-source code review pushes the boundaries of how quickly information can spread in code review: On average, information can spread through the code review system of Visual Studio Code in less than a day and at React less than three days. The average (median) temporal distance between two code review participants at Android, Trivago, and Spotify is less than seven days, while a code review participant a Microsoft takes more than 14 days, which is still in the observation window of four weeks.

From Figure 4.6, we observe that the temporal distances in closed-source code review systems are tightly constrained to a typical workday schedule. All closed-source code review systems exhibit distinct step patterns in their ECDFs, reflecting a pronounced day-night rhythm. In contrast, the ECDFs for all open-source code review systems are notably smoother, suggesting a more continuous and globally distributed review activity with less coupling to traditional workday boundaries.

4.5 Discussion

As outlined in the experimental design in Section 4.3, we now turn to a discussion of the results as evidence testing the theory that code review would function effectively as a communication network. In other words, do our four measurements indicate that code review fails to enable information diffusion at scale, thereby calling into

[14]In the replicated study, we inadvertently reported Microsoft's topological distances with an off-by-one error. Although we consider this a minor issue that does not impact the validity of the previous findings, we believe it is important to explicitly acknowledge the error. We have verified the results and corrected it in this paper to ensure completeness and accuracy.

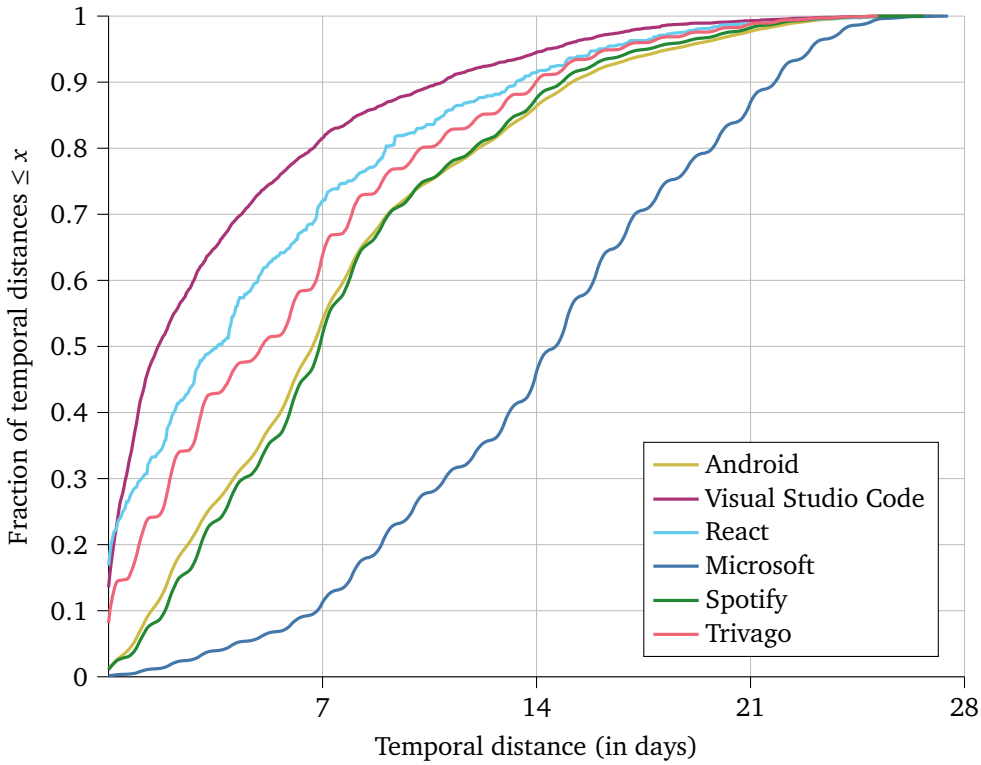


Figure 4.6: The cumulative distribution of minimal temporal distances between participants in code review systems. The temporal distance is the minimal duration required to spread information from one participant to another.

question its role as an effective communication medium for information exchange among developers?

Figure 4.7 summarizes and highlights the interdependencies among the four measurements, the two research questions, and the two propositions introduced in Section 4.3.

4.5.1 Code review can spread information widely among its participants (Proposition 1)

Our results provide strong support for the first proposition, particularly in closed-source environments. Participants in code review at Trivago or Spotify, for example, can reach up to 85% of their network under best-case conditions (cf. Figure 4.4), and a large proportion of participants achieve high diffusion ranges. These results reflect dense, tightly integrated communication structures that promote high reachability. In terms of absolute information diffusion range, the closed-source system at Microsoft stands out with participants reaching up to 26,216 others, even though its normalized range is relatively low due to its sheer scale (cf. Table 4.6).

However, this capability does not translate uniformly across environments. Our simulation revealed that open-source code review systems consistently fall short of the upper bounds set by closed-source systems in terms of how widely information can spread (cf. Figure 4.8). Even popular and highly relevant open-source projects such as React and Android only approach but do not exceed the normalized diffusion range seen in closed-source systems.

Our results also identified three distinct diffusion patterns across systems (cf. Figure 4.4), yet no simple variable (e.g., system size, tooling, or openness) explained these differences. For instance, the median reach in Android is less than 1,500 participants, while Spotify's median exceeds this, despite similar number of participants (cf. Table 4.6). These results suggest that structural factors—such as internal communication protocols or enforced participation policies—may facilitate wider diffusion in closed-source environments.

Thus, while H_1 is supported overall, the degree of diffusion is strongly modulated by whether the system is open or closed source. Closed-source code review systems consistently enable wider diffusion. We speculate that the underlying organizational dynamics play a critical role in shaping these outcomes. In particular, closed-source systems may support broader information diffusion because they tend to operate in more decentralized review environments, where code review responsibilities are distributed across multiple engineers and teams. In contrast, open-source projects often centralize review activity in a small set of core maintainers, who act as gatekeepers and conduct the majority of reviews. This centralized control, combined with limited social or organizational trust toward transient or unknown contributors, may constrain the pathways through which information can spread.

Figure 4.7: Summary of the four information diffusion measurements used in the study, their alignment with the research questions RQ 1 and RQ 2, and references to the corresponding propositions.

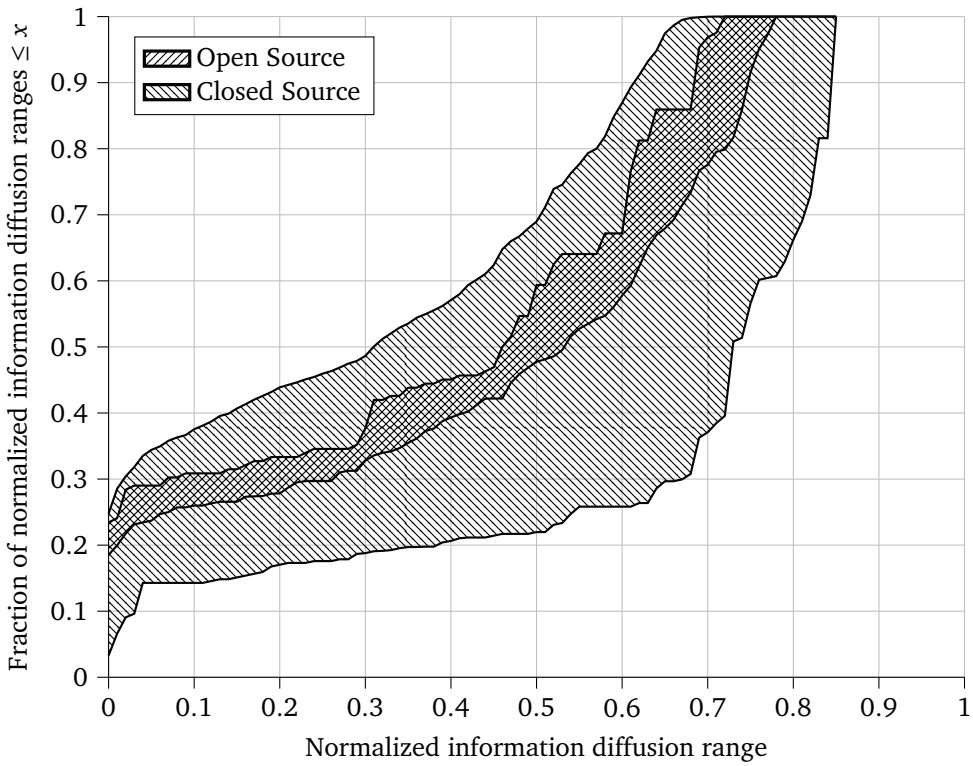


Figure 4.8: The aggregated empirical cumulative distributions of the normalized information diffusion ranges in open-source and closed-source code review systems after four weeks.

4.5.2 Code review can spread information quickly among its participants (Proposition P_2)

The results also support proposition P_2: information can spread quickly among code review participants, particularly in open-source environments. In terms of topological distance, which measures the number of hops between participants, open-source systems such as React and Visual Studio Code define the shortest communication paths across our sample (cf. Figure 4.5). In React, the average path length between any two participants is just one hop, indicating high connectivity and short transmission chains.

These structural observations are reinforced by measurements of temporal distance—how long it takes for information to spread along time-respecting paths. Again, React sets the benchmark: on average, information can reach any participant in less than 3.5 days (cf. Figure 4.6). In contrast, Microsoft's system—despite its large reach—exhibits the longest average delay (over 14 days), suggesting that scale may come at the cost of immediacy.

Temporal activity patterns further differentiate the systems. Open-source code reviews exhibit smoother ECDFs of temporal distance, with less evidence of the day-night cycles seen in closed-source systems (cf. Figure 4.6). This likely reflects the distributed and asynchronous nature of open-source contributions, where global participation enables continuous communication, independent of localized work schedules.

Together, the topological and temporal distance results demonstrate that, under idealized conditions, information can not only spread quickly through the code review network but may do so more efficiently in decentralized and open environments. At the same time, our findings also reinforce our previous speculation: that centralized review structures—common in open-source projects—may contribute to shorter topological paths by concentrating review activity through a small number of highly active core maintainers. These maintainers effectively act as a communication hubs, reducing the number of hops needed for information to flow between otherwise unconnected participants. While this centralized structure may constrain the overall diffusion range, it can facilitate rapid spread within a smaller core network. In contrast, closed-source systems often distribute review responsibilities more broadly across teams, which supports wider diffusion but can result in longer average paths. Future work should explore this trade-off more systematically, by empirically quantifying the degree of review centralization and its relationship to both diffusion speed and reach across diverse organizational and community contexts.

4.6 Implications

In this section, we explore the implications and broader value of conceptualizing code review as a communication network that is grounded in a stronger theoretical

foundation. Our aim is to show how a well-supported theory can function not only as an analytical tool, but also as a vehicle for advancing our understanding of collaborative software engineering. While many implications may follow from this perspective, we highlight three that we believe are particularly significant—for research and for practice—each demonstrating how this theory can inform future directions in software engineering.

4.6.1 Code Review as Proxy for Collaborative Software Engineering

If code review functions as a communication network among developers it can serve as a measurable and reliable proxy for the—potentially very fine-grained—collaboration. While insights into collaborative software engineering have wide-ranging applications in research and practice, the collaboration within a multinational enterprise across national borders has an often overlooked legal implication: the profits from those cross-border collaborations within a multinational enterprise become taxable [37]. Given that tax compliance presents a significant financial risk to multinational enterprises—as highlighted by the ongoing case involving Microsoft [124]—it is imperative that multinational enterprises rely on robust and reliable measurements, grounded in a solid theoretical framework, to ensure accurate reporting to tax authorities and to mitigate the risks associated with non-compliance. In our work [37], we suggest cross-border code reviews—instances where participants are employed by legal entities in different countries—at large multinational as proxy for collaborative software engineering across international borders and lay the foundation for a new line of research on tax compliance in collaborative software engineering [37].

4.6.2 Differences Between Open-Source and Closed-Source Code Review

Our findings reveal notable differences in how open-source and closed-source code review systems function as a communication networks. These differences are not merely a matter of scale, but reflect deeper variations in organizational structure, governance, and review practices. Open-source projects often exhibit more centralized review activity around a small group of maintainers [1, 17, 98], while closed-source systems distribute review responsibilities more broadly across formal teams with shared accountability [106, 110]. These distinct patterns could explain the substantial differences in the capabilities of code review as a communication network in open-source and closed source software development.

As a result, care must be taken when interpreting empirical findings across these contexts. Many prior studies rely on open-source data—often due to its accessibility and transparency—but our results suggest that insights gained from open-source systems may not generalize to closed-source environments without further considerations. Researchers and practitioners should be cautious in transferring assumptions or models derived from one setting to another without considering structural differences.

But rather than viewing these differences as limitations, we see them as opportunities to develop a more nuanced understanding of code review as a socio-technical process. The apparent strengths of both open-source (speed) and closed-source (reach) environments suggest the potential value of hybrid models like inner source [21].

4.6.3 Impact of AI on the Communicative Role of Code Review

A more forward-looking implication of our work concerns the evolving role of code review in light of rapid advances in AI-assisted development. As automated tools increasingly contribute to code generation, validation, and even code review, the practice of code review is likely to undergo substantial transformation.

While deterministic bots have long played a role in code review—for example, in reporting testing results or code quality characteristics—the emergence of non-deterministic, generative AI systems introduces a qualitatively different shift. Unlike rule-based tools, generative AI can produce context-sensitive and novel responses, potentially functioning as autonomous participants in code review, both as reviewer or code author. As these systems might transition from assistants to co-reviewers or even sole reviewers, they may begin to reshape the communicative and collaborative function of code review itself.

We anticipate three key types of erosion that may ultimately signal the end of code review as we know it today. First, *erosion of understanding*: AI-generated code and reviews may prioritize machine comprehension over human readability, reducing developers' ability to maintain and reason about the codebase. Second, *erosion of accountability*: as AI systems cannot be held responsible for the changes they influence, gaps in ownership and regulatory compliance may emerge. Third, *erosion of trust*: the blurring of authorship between humans and AI risks undermining confidence in the authenticity and reliability of the review process.

Together, these concerns suggest that deeper integration of generative AI into code review may not simply improve efficiency but fundamentally alter—or even dismantle—the collaborative, human-centered foundation of code review. Without careful oversight, we may need to radically rethink its role in future software development ecosystems, both in research and practice.

4.7 Threats to Validity

As with any empirical study, our findings are subject to limitations that may influence the strength, scope, or interpretation of our results. To support transparency and enable critical assessment, we reflect on the potential threats to the validity of our study, following established frameworks in empirical software engineering. Specifically, we discuss threats to internal validity (whether our results are caused by the study

design rather than confounding factors), construct validity (whether our measurements accurately reflect the concept of information diffusion), external validity (the extent to which our findings generalize to other settings), and conclusion validity (the reliability and potential bias in our inferences and interpretations). Where possible, we outline strategies to mitigate these threats and suggest directions for future work to strengthen the robustness of the research.

4.7.1 Internal Validity

Our study simulates information diffusion under best-case conditions. However, several factors may threaten internal validity—that is, whether the observed outcomes can be confidently attributed to our simulation logic rather than to uncontrolled or confounding variables.

First, the boundary between open-source and closed-source development introduces a structural asymmetry that may affect internal consistency. In closed-source environments, the development context is clearly bounded: contributors are typically employees, and communication occurs within the confines of an organization. In contrast, open-source projects often lack a well-defined membership boundary. Developers may participate across multiple projects, both professionally and voluntarily, and information frequently flows between interrelated codebases. To ensure a consistent analytical scope, we rely on the boundaries defined by the code review tool and its configuration. While technically consistent, this boundary may be artificial—particularly in open-source settings where review activity and knowledge sharing extend across repositories. As a result, our simulation may underestimate the actual diffusion potential in open ecosystems. Future work could address this limitation by modelling open-source environments as interlinked review networks rather than isolated systems.

Second, a structural limitation arises from the intrinsic differences in scale between open-source and closed-source code review systems. Open-source projects, by nature, rarely reach the scale of corporate systems. Even the largest open-source systems in our sample (e.g., Android or Visual Studio Code) involve only a few hundred active participants, whereas Microsoft's internal review network includes over 37,000 contributors (cf. Table 4.3). These differences are not incidental but reflect fundamental constraints of each ecosystem. Consequently, comparing diffusion capacity across systems of such disparate scale risks conflating size with capability. Closed-source systems are simply able to support broader diffusion because they operate at a fundamentally different scale. Even with normalized metrics, the raw potential for information to reach more participants in large systems may amplify or mask structural effects that are not present in smaller, open networks. Thus, any comparison of diffusion between open-source and closed-source systems must be interpreted with caution: what appears to be a performance difference may instead reflect the structural ceiling each system type can naturally support.

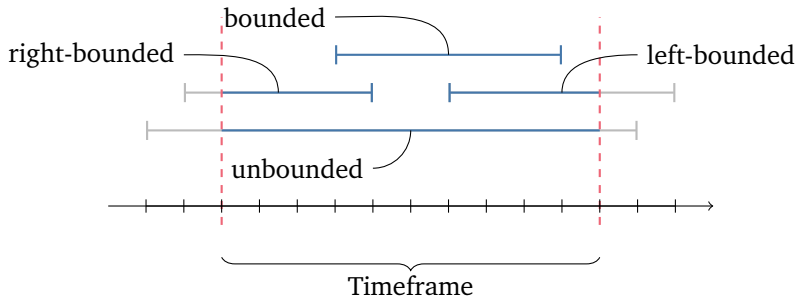


Figure 4.9: The impact of the timeframe on data completeness: The concurrent code reviews as a communication channels may have started before or ended after the observed timeframe. Due to the cut, the communication channels may cut at their start (right-bound) or at their end (left-bound), or the channel is completely contained (bound) or not contained (unbound) in the timeframe.

Finally, the finite timeframe of our simulation introduces unavoidable edge effects. Information diffusion is a continuous process, but our measurement window imposes artificial start and end points. Some communication channels may begin before or continue after the observation period (Figure 4.9). Figure 4.9 shows the distribution of communication channels classified by their temporal boundaries across the studied code review systems: bounded, left-bounded, right-bounded, and unbounded. While the proportions vary between all systems, the share of problematic right-bounded and unbounded channels—where code review participants may be missed and information diffusion potentially underestimated—is consistently small for Android, Visual Studio Code, React, and Spotify (less than about 10%). In contrast, these edge effects are more pronounced in the code review systems at Microsoft and Trivago, suggesting that we may underestimate the upper bound of information diffusion in those systems. We found no clear explanation for this pattern within the observed timeframe.

4.7.2 Construct Validity

Our study operationalizes information diffusion using structural metrics derived from simulated code review interactions. These measurements estimate how widely and quickly information could diffuse through code review networks under idealized, best-case assumptions: namely, that information spreads whenever structural and temporal conditions permit. While this approach provides a useful upper bound on diffusion potential, it likely overstates the extent of real-world communication, which depends on human factors such as selective attention, comprehension, memory, and intent. As a result, our structural measurements may partially misalign with the theoretical construct of code review as a human-centered communication network.

In particular, we treat review participation as a proxy for meaningful communication, without accounting for variation in message quality, reviewer influence, or interaction

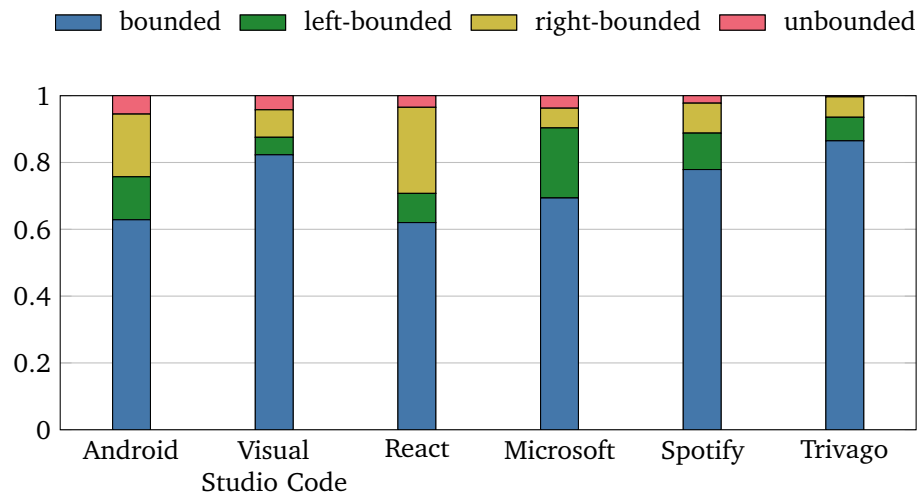


Figure 4.10: Shares of bounded, left-bounded, right-bounded, and unbounded communication channels among all code review systems within the simulation timeframe.

depth. This abstraction overlooks important cognitive and social processes that govern actual information exchange—such as whether reviewers internalize feedback, whether authors incorporate it, and how review discussions are shaped by trust or reputation. These simplifications pose a threat to construct validity, as our metrics may capture structural reach rather than substantive communication.

A further threat arises from the potential inclusion of automated accounts or partially automated review behaviors in our analysis. To preserve the validity of our measurements as proxies for human communication, we aimed to exclude bot activity from our data. This filtering might be more effective in open-source systems, where public metadata and transparent contribution histories enabled fine-grained identification and removal of automated accounts. In closed-source environments, however, access restrictions and anonymized user identifiers limited our ability to reliably detect non-human actors. As a result, instances of developer-driven automation—such as scripted review approvals, preconfigured responses, or notification handlers—may have remained in the dataset. These interactions do not reflect meaningful human engagement and could therefore distort our diffusion measurements. The inclusion of such automation introduces noise into our structural metrics and risks weakening their alignment with the underlying construct of human-to-human information flow in code review.

To improve construct validity in future work, researchers could incorporate behavioral constraints or cognitive models of information processing into simulation logic. For

example, empirically derived probabilities of message retention, forwarding, or selective attention—based on factors like review length, comment type, reviewer load, or past collaboration—could bring modelled diffusion closer to real developer behavior. Additionally, refining the operationalization of diffusion through multi-dimensional metrics that integrate structural paths with semantic indicators (e.g., comment sentiment, topic alignment, or question–response dynamics) may yield a richer and more accurate view of how knowledge spreads through code review.

Finally, validating structural diffusion measures against ground truth—such as developer-reported awareness, trace links to bug resolution, or interview data—could help assess whether our metrics truly reflect meaningful communication outcomes. By combining structural models with empirical insights into how developers communicate and learn during code review, future studies can enhance both the realism and theoretical alignment of information diffusion as a construct.

4.7.3 External Validity

As discussed in Section 4.2.4, comparing open-source with closed-source software development is inherently difficult since the definition of open source refers only to a specific type of licensing but not to a specific software development process. The open-source projects included in our study—React (maintained by Meta), Android (led by Google), and Visual Studio Code (maintained by Microsoft)—represent a specific subset of the open-source landscape: large, industry-backed initiatives. Although these projects formally meet the definition of open source, they also benefit from substantial institutional support—such as dedicated engineering staff, structured review processes, and sustained corporate oversight. As a result, they occupy a hybrid space that merges elements of community-driven open source with characteristics of enterprise-level software engineering. This hybrid model sets them apart from both traditional closed-source systems and purely community-led open-source projects. While the inclusion of these corporate-backed initiatives provides valuable insight into scalable, production-grade review practices, it also limits the generalizability of our findings to other open-source contexts—particularly those that are grassroots, volunteer-driven, or lacking institutional infrastructure.

Future research could examine whether the communication and diffusion dynamics observed in industry-backed open-source systems extend to smaller, community-driven open-source projects. These projects may operate under different social norms, resource constraints, and review practices, which could significantly influence how information spreads through their code review networks. Expanding the empirical base to include a broader spectrum of open-source ecosystems is therefore essential to improving the external validity of studies comparing open- and closed-source development.

4.7.4 Conclusion Validity

Like any empirical study, our work is subject to potential research bias—particularly given its nature as a replication study involving an overlap with the original research team. While the experimental design with its explicit modelling of assumptions and quantitative approach may reduce certain forms of subjectivity compared to qualitative methods, the interpretation of results still depends on the perspectives and decisions of the researchers involved. This risk is especially relevant in self-replications, where unconscious bias may influence framing, emphasis, or analytical choices. To mitigate this threat, we have made all data and simulation materials publicly available and invite the research community to independently replicate our findings, extend them to other contexts, or challenge our conclusions. In doing so, we hope to contribute to a cumulative body of evidence—a family of replications—that collectively strengthens the empirical foundation of code review as a communication network [23].

4.8 Conclusion

Our study provides a first confirmatory perspective on the theory of code review as a communication network. First, we found that code review networks can indeed spread information widely and quickly, supporting the theory of code review as a communication acorrding to which information can spread widely and quickly through code review.

Second, we found no evidence that larger network size impairs information diffusion. Even within one of the largest code review systems worldwide—Microsoft's internal code review—information still can spread relatively widely and quickly. This suggests that, at least under best-case conditions, code review networks scale without a loss in their capacity to diffuse information. However, this capability is not guaranteed; the ability of code review systems to support information diffusion is not uniformly distributed across our sample. This underscores that effective information diffusion is not an inherent trait of code review, but a property that must be intentionally cultivated through structure, practice, and design.

Third, our findings reveal important differences between open-source and closed-source code review systems with regards to the capability of information diffusion. While open-source projects tend to spread information more quickly, they do so across a significantly smaller fraction of participants compared to their closed-source counterparts. This suggests that findings derived from studies of open-source code review may not necessarily generalize to closed-source environments. Given these structural and behavioral differences, we advocate for a critical reevaluation of the generalizations made in the software engineering research community based primarily on open-source code review.

Taken together, our results both reinforce and refine the theory of code review as a communication network, and they highlight important contextual boundaries that future research and practice must take into account. By formalizing and validating the theory of code review as a communication network, we contributed to a deeper understanding of how code review function and to theory-driven and theory-oriented software engineering [115].

Data Availability

The replication package, including simulation code, datasets, and analysis scripts, is publicly available at

`https://github.com/michaeldorner/capability-of-code-review-as-communication-network`.

Please note that the complete replication package will be archived on Zenodo following the completion of the review process.

Acknowledgments

This work was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

Measuring Information Diffusion in Code Review

5

This chapter is based on Michael Dorner, Daniel Mendez, Ehsan Zabardast, Nicole Valdez, and Marcin Floryan. "*Measuring Information Diffusion in Code Review at Spotify*". Registered report accepted at *International Symposium on Empirical Software Engineering and Measurement (ESEM)*; manuscript currently under review at *Journal of Empirical Software Engineering*. 2025.

Abstract

Background: Code review, a core practice in software engineering, has been widely studied as a collaborative process, with prior work suggesting it functions as a communication network. Despite its popularity, this theory has not been formalized and remains untested, limiting its practical and theoretical significance.

Objective: This study aims to (1) formalize the theory of code review as a communication network explicit and (2) empirically test its validity by quantifying the extent of information diffusion—the spread of information—in code review across social, organizational, and software architectural boundaries.

Method: We conduct a large-scale empirical analysis of 220,733 code reviews by 2,246 developers at Spotify during 2019. We operationalize information diffusion through three lenses: participant dissimilarity across reviews (social), cross-team developer involvement (organizational), and architectural linkage of reviewed components.

Results: We find that over 99.6% of review pairs have completely distinct participant sets, indicating high diffusion across social boundaries. Approximately 18% of code reviews involve developers from multiple teams, evidencing nontrivial diffusion across organizational boundaries. Of the 5.82% of code reviews linked to others, 99.0% span distinct repositories, reflecting architectural diffusion.

Conclusion: The substantial diffusion of information across social, organizational, and architectural boundaries empirically supports the theory of code review as a communication network. These findings indicate that code review plays a role not only in quality assurance, but also in enabling communication and coordination in large-scale, distributed software projects. They further support its use as a measurable proxy for cross-border collaboration in the context of tax compliance, but also raise concerns about the impact of integrating LLMs on its communicative function.

5.1 Introduction

Modern software systems are often too large, too complex, and evolve too fast for an individual developer to oversee all parts of the software and, thus, to understand all implications of a change. Therefore, most collaborative software projects rely on code review to foster informal and asynchronous discussions on changes and their impacts before they are merged into the code bases. During those discussions, participants exchange information about the proposed changes and their implications, forming a communication network that emerges through code review.

This perspective on code review as a communication network is supported by a broad body of prior qualitative research. As a core practice in collaborative software engineering, the communicative and collaborative nature of code review has been examined in various studies [7, 14, 17, 29, 106]. Our synthesis of this prior research

revealed a consistent pattern: practitioners widely perceive code review as a communication medium for information exchange, effectively functioning as a communication network [39]. This recurring pattern provides a solid foundation for the emerging *theory of code review as a communication network*, which conceptualizes code review as a process that enables participants to exchange information around a code change.

However, the theory—as often in software engineering research [115]—remains implicit and has yet to be formalized. Furthermore, relying solely on this mostly exploratory and qualitative research to ground our understanding of code review as a communication network falls short. Exploratory research begins with specific observations, distills patterns in those observations, typically in the form of hypotheses, and derives theories from the observed patterns through (inductive) reasoning. The nature of exploratory and especially qualitative research enables to analyze chosen cases and their contexts in great detail. However, their level of generalizability is also limited as they are drawn from those specific cases (especially when there is little to no relation to existing evidence that would allow for generalization by analogy). Therefore, while such initial theories may well serve as a very valuable foundation, they still require multiple confirmatory tests as, otherwise, their robustness (and scope) remains uncertain rendering it difficult to establish credibility, apply in practice, or develop it further. We thus postulate that exploratory (inductive) research alone is not sufficient to achieve a strong level of evidence, as discussed also in greater detail by Wohlin [128]. Confirmatory research, in turn, typically forms a set of predictions based on a theory (often in the form of hypotheses) and validates to which extent those predictions hold true or not against empirical observations (thus testing the consequences of a theory). To efficiently build a robust body of knowledge, we need both exploratory and confirmatory research to minimize bias and maximize the validity and reliability of our theories efficiently. Figure 5.1 shows this interdependency between exploratory (theory-generating) and confirmatory (theory-testing) research in empirical research.

In this context, we set out to address a gap by formalizing and empirically validating the theory of code review as a communication network through the capability of code review to diffuse information across boundaries. From this perspective, we propose the following theory: code review, as a communication network, enables information to spread across

1. participants (i.e., social boundaries),
2. teams (i.e., organizational boundaries), and
3. software components (i.e., architectural boundaries).

Whether this theory holds depends on the actual capabilities of code review systems to facilitate such information diffusion across those different boundaries. If a given code review system exhibits little or no diffusion across those boundaries at all, it would

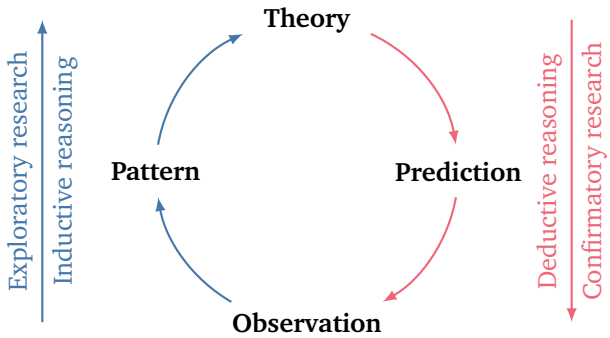


Figure 5.1: The empirical research cycle (in analogy to [77]): While exploratory research is theory-generating using inductive reasoning (starting with observations), confirmatory research is theory-testing using deductive reasoning (starting with a theory). The research at hand is confirmatory.

challenge the general applicability of the theory to practical terms, suggesting that additional constraints, contextual factors, or boundary conditions must be considered to explain under which conditions and how code review functions effectively as a communication network.

Thus, our goal is not to confirm or reject a null hypothesis, but to evaluate whether the empirical patterns observed are consistent with—or challenge—the theoretical view of code review as a communication network. A system showing minimal diffusion may indicate that the theory, in its current form, lacks explanatory power or universality.

To that end, we measure information diffusion in code review at Spotify across social, organizational, and architectural boundaries and address the following research questions:

- RQ_1: To what extent does code review enable information diffusion across *social* boundaries between developers?
- RQ_2: To what extent does code review enable information diffusion across *organizational* boundaries between teams?
- RQ_3: To what extent does code review enable information diffusion across *architectural* boundaries between software components?

These research questions operationalize the core expectations of our theory of code review as a communication network. If, for instance, RQ_1 reveals that the same participants recur frequently across code reviews, RQ_2 shows that most reviews involve only a single team, or RQ_3 finds that linked reviews rarely cross component boundaries, this would challenge the notion that code review facilitates broad information diffusion. Conversely, consistent evidence of diverse participation, cross-team collaboration, and cross-component linkage would provide empirical support for the theory.

The contributions of our research are twofold: First, we formalize the theory of code review as a communication network, advancing a theory-driven perspective in software engineering research [115]. Second, we empirically evaluate this theory by quantifying the extent of information diffusion in code review, thereby extending the findings of Dorner et al., which approximate the upper bound of information diffusion through a simulation.

Throughout this study, we are following the *International Vocabulary of Metrology* [65]. In particular, we adopt the definitions of measurement model, measuring system, and measurement, as well as their relationships.

The remainder of this study is structured as follows: After we discuss the related work in Section 5.2, we describe the research design in Section 1.3 in which we derive the three propositions and describe the measurement model, the measuring system, and the measurement at Spotify following the *International Vocabulary of Metrology* [65]. Since the discussion of the results for evaluating our theory is at the heart of this study, we outline of our discussion in Section 5.6. We close the registered report by discussing the threats to validity as known before seeing the data in Section 5.7.

5.2 Background

To clarify our theoretical perspective and situate our work within a broader scientific context, we first outline our view on the role of theories in software engineering research. We then review prior work on measuring communication in code review, which informs our operationalization. The prior research that forms the basis for our theory of code review as a communication network is discussed in Section 5.3.1.

5.2.1 Theories in Software Engineering

Scientific theories[1] are arguably the demarcation line behind which we may well see our discipline as an insight-oriented, scientific one. In fact, theories are the core of empirical software engineering and the community recognizes the importance of "theor[ies] as both a driver for and a result of empirical research in software engineering" [115]. This view aligns well with the position of Stol and Fitzgerald, who argue that theories should not be treated as an optional component or post-hoc reflection, but as a central element of scientific inquiry in our field. This attitude seems to also reflect well the general attention given by the software engineering community to the need for more evidence-based, empirical research, thus, turning what was once an engineering discipline governed by rationalist arguments, folklore and conventional wisdom towards what can today be seen as a paradigmatic stage of normal science [75, 77]. At the same time, our discipline is challenged by various

[1] We characterize a scientific theory as the belief that there are patterns in phenomena while having survived (1) tests against sensory experiences and (2) criticism by critical peers [75]

factors rendering theory building and theory evaluation often cumbersome, and leaving many theoretical implications of research results at an implicit state.

Following this perspective, we therefore view theory not only as a lens through which to interpret empirical findings, but also as an explicit outcome that emerges from systematically studying software engineering phenomena. In this study at hands, we apply this thinking by first conceptualizing code review as a communication network based on previous qualitative studies. Rather than focusing solely on technical artifacts or code review outcomes, we examine the structural and relational aspects of code review interactions as a means to understand its contribution to collaborative software engineering. Our goal is to contribute to increasing the robustness of the theory by offering falsifiable, empirically grounded explanations that others in the community can further test and refine in the future. This is also well aligned with the core idea that scientific progress should have driven by the formulation of bold conjectures and rigorous, independent attempts at falsification [94]. We adopt this view by treating our theory not as a set of universal truths, but as a set of provisional explanations that can and should be tested against empirical data for us to better understand the various context factors under which a theory may or may not hold true. From this position, our conceptualization of code review as a communication network is intended to generate first falsifiable insights, contributing to the iterative refinement of theory in software engineering in the long run.

5.2.2 Measuring Communication in Code Review

Although different qualitative studies report information sharing as a key expectation towards code review [7, 14, 17, 29, 106], only three prior studies have quantified information exchange in code review.

In an *in-silico* experiment, we simulated an artificial information diffusion within large (Microsoft), mid-sized (Spotify), and small code review systems (Trivago) modelled as communication networks [39]. We measured the minimal topological and temporal distances between the participants to quantify how far and how fast information can spread in code review. In this interventional study, we found evidence that the communication network emerging from code review scales well and spreads information fast and broadly, corroborating the findings of prior qualitative work. The reported upper bound of information diffusion, however, describes information diffusion in code review under best-case assumptions, which are unlikely to be achieved in practice. While the upper bound of information diffusion provides insight into the potential of code review as a communication network under idealized conditions, the present study complements this prior simulation with a non-interventional, observational perspective [6]— drawing on real-world code review systems to assess how much information diffusion actually occurs in practice.

In the first observational study, [99] extended the expertise measure proposed by Mockus and Herbsleb. The study contrasts the number of files a developer has modified with the number of files the developer knows about (submitted files ∪ reviewed files) and found a substantial increase in the number of files a developer knows about exclusively through code review.

A second observational study [106] reports (a) the number of comments per change a change author receives over tenure at Google and (b) the median number of files edited, reviewed, and both—as suggested by Rigby and Bird. The study finds that the more senior a code change author is, the fewer code comments he or she gets. The authors "postulate that this decrease in commenting results from reviewers needing to ask fewer questions as they build familiarity with the codebase and corroborates the hypothesis that the educational aspect of code review may pay off over time." In its second measurement, the study reproduces the measurements of Rigby and Bird but reports it over the tenure of employees at Google. They showed that reviewed and edited files are distinct sets to a large degree.

Although the proposed file-based network creation is a sophisticated approach and may serve as a complement measurement in future studies, we found the following limitations in the measurement applied in prior work:

- File names may change over time, which introduces an unknown error to those measurements.
- The software-architectural or other technical aspects (e.g., programming language, coding guidelines) of code make the measurements difficult to compare in heterogeneous software projects.
- We are unaware of empirical evidence that passive exposure to files in code review would lead to improved developer fluency.
- The explanatory power of both measurements is limited since the authors set arbitrary boundaries: Rigby and Bird excluded changes and reviews that contain more than ten files, and Sadowski et al. limited the tenure of developers to 18 months and aggregated the tenure of developers by three months.

Furthermore, our code-review-based approach differs in two aspects: First, information in code review is not only encoded in the source code but also is also in the discussions within a code review. A file-based approach does not reveal this type of information diffusion. Our code-review-based approach includes information encoded in the affected files and in the related discussions but also subsumes information on other abstraction layers of the software system. Second, a file-based approach assumes a passive and implicit information diffusion. That is, information is passively absorbed during review by the developers. In contrast, the information diffusion captured by a code-review-based approach like ours is an active information diffusion,

that is, a developer actively and explicitly links information that she or he deems to be worth linking, which makes linking a human, explicit, and active decision.

5.3 Theory of Code Review as a Communication Network

This section formalizes our theory of code review as a communication network.

5.3.1 Foundations in Exploratory Research

The theory of code review as a communication network builds on a body of exploratory studies that investigate the motivations for and expectations toward code review in industrial settings [7, 14, 17, 29, 106]. Synthesizing this literature, Dorner et al. identified information[2] exchange as the common cause behind these observed effects of code review [39]. Although existing work has described the collaborative nature of code review, it has not formalized a theory of code review as a communication network for its participants.

5.3.2 Theoretical Model

Our research is grounded in the theory that code review functions as a communication network which has the capability to spread information across different contexts that would otherwise remain disconnected. From this perspective, code review is not only a quality assurance mechanism but also a process through which information diffuses across boundaries. We refer to this phenomenon as *information diffusion*. In this study, we examine the extent to which code review supports information diffusion across three types of boundaries:

- *Social boundaries*, defined by informal participant structures.
- *Organizational boundaries*, defined by formal team structures.
- *Software architectural boundaries*, defined by the modular structure of the codebase.

These boundaries are not explicitly defined within code review but become observable through participation patterns. For instance, if developers from multiple teams collaborate on the same code review, an organizational boundary has likely been

[2]We use the term information rather than knowledge, following the reasoning of Dorner et al. and Pascarella et al., even though earlier work used the latter term Bacchelli and Bird, Baum et al., Bosu et al., Cunha, Conte, and Gadelha, Sadowski et al. Although not synonymous, information can be seen as a superset of knowledge: knowledge is meaning derived from information through interpretation. Not all information constitutes knowledge, but all knowledge originates as information. This framing allows us to subsume differing notions of knowledge without engaging in epistemological debates or addressing truth claims often associated with knowledge. Code review may include subjective elements such as opinions, assumptions, or misunderstandings—forms of information that may not meet all definitions of knowledge or truth.

crossed. If the same developers appear across many different code reviews, this suggests interaction across social boundaries. Likewise, if reviews are linked across different software components, this indicates diffusion across architectural boundaries.

The capability of code review to cross these boundaries is the defining characteristic of its function as a communication network. Our theoretical model posits that this boundary-crossing behavior is indicative of information diffusion and can be observed through empirical data.

5.3.3 Hypotheses

From the theoretical model, we derive three testable hypotheses, each corresponding to a type of boundary:

- H_1: Code review enables the diffusion of information across social boundaries between developers.
- H_2: Code review enables the diffusion of information across organizational boundaries between teams.
- H_3: Code review enables the diffusion of information across architectural boundaries between software components.

These three hypotheses are logically tied to our overarching theory, which posits that code review functions as a communication network. We express this relationship in propositional logic as:

$$T \Rightarrow (H_1 \wedge H_2 \wedge H_3) \tag{5.1}$$

where T denotes the theory that code review functions as a communication network. This formulation implies that the theory can only be fully corroborated in its general form if, and only if, all of the hypotheses are supported by empirical evidence. Rather than relying on arbitrary statistical thresholds, we interpret the presence or absence of diffusion based on the observed patterns of code review at Spotify.

5.4 Research Design

To empirically evaluate our theory that code review functions as a communication network enabling cross-boundary information diffusion, we conduct a confirmatory, observational study [6]. This section describes the research context, the data collection, the operationalization of theoretical constructs, and their corresponding measurement procedures.

5.4.1 Research Context

This study is situated within Spotify, a large software-intensive organization that develops and maintains a modular codebase composed of thousands components, most of them encapsulated in its own repository and maintained by largely autonomous teams [110]. Code review is a very common and standardized practice, supported by developer tools—in this case GitHub—and fully integrated into the trunk-based development process.

Developers at Spotify may be affiliated with multiple teams simultaneously, often as part of a practice known as *embedding* [110]. In embedding, engineers temporarily join another team, typically for several months—to gain new competencies or to support projects requiring additional resources. These embedded roles, initiated by individual developers or teams and approved by engineering managers, result in a dynamic team structure with overlapping affiliations.

What makes Spotify particularly valuable as a research setting is not only the scale and organizational characteristics but also the quality and completeness of its metadata. In addition to comprehensive code review meta data collected from GitHub Enterprise, we had access to detailed, time-resolved information about team affiliations and developer identities through internal systems. This metadata enables precise tracking of social and organizational boundaries, enhancing the interpretability and reliability of our measurements. Taken together, these factors make Spotify a uniquely rich empirical context for testing our theory of code review as a communication network.

5.4.2 Data Collection

To collect the necessary data for testing our theory, we relied on two primary data sources: GitHub Enterprise for extracting code review meta data, and an internal Spotify system for retrieving developers' team affiliations.

To extract all internal code review at Spotify, we used GitHub's REST API. In GitHub's data model, a code review is represented as a special type of issue (i.e., a pull request). This allows access to the full history of each code review via the timeline events endpoint of the `Issues` API,[3] which records all activity within a pull request—including comments, status changes, and references to other pull requests. Using the endpoints for repositories and pull requests, we collected all available timeline events for every pull request across all internal repositories hosted on Spotify's GitHub Enterprise instance.[4] We focused on all events recorded during the 2019 calendar year. This timeframe was selected to enable the full publication of the anonymized dataset while minimizing potential conflicts with ongoing development and privacy concerns.

[3] `https://docs.github.com/en/enterprise-server@3.10/rest/issues/timeline?apiVersion=2022-11-28#list-timeline-events-for-an-issue`

[4] At the time of writing, GitHub's general event endpoint `/events` is limited to the 300 most recent events from the past 90 days, making it unsuitable for extracting historical data.

To determine team affiliations, we used an internal system at Spotify that captures daily snapshots of each developer's organizational membership. This allowed us to assign team affiliations with day-level accuracy for all developers, even though the data pertains to the year 2019.

To ensure that our dataset captures meaningful human participation in code review, we applied strict filtering: We excluded all interactions performed by automated accounts or bots, which we identified as GitHub accounts not present in the internal organizational system, and retained only events representing actual review activity, specifically the creation, commenting, closing, merging, or linking of pull requests. Other one-click interactions—such as likes, assignments, or subscriptions—were considered minimally meaningful and therefore excluded from the dataset.

After filtering, our dataset comprises 220,733 code reviews conducted by 2,246 human developers in the year 2019. Among these, 15,075 code reviews are linked to at least one other. For 0.02% of all code reviews, we identified human code review participants for whom team affiliation could not be reliably determined, resulting in code reviews with no team affiliation at all. We discuss the implications and limitations of this exclusion in the corresponding result (Section 5.5) and discussion sections (Section 5.6). The code reviews in our sample timeframe belong to 1,786 different repositories.

To summarize, the dataset captures all internal and human code review activity at Spotify in 2019, offering a comprehensive foundation for analyzing the information diffusion in a large-scale industrial software organization.

5.4.3 Operationalization and Measurement Procedures

To empirically test the theory that code review functions as a communication network enabling information diffusion across boundaries, we operationalize the three theoretical constructs (information diffusion across social, organizational, and architectural boundaries) through a specific modelling approach and a corresponding measurement procedure, grounded in structural characteristics observable in code review metadata.

We conceptualize information diffusion as the observable crossing of contextual boundaries through code review activity. Each type of boundary is modelled with respect to a different unit of analysis: reviewer participation, team affiliation, and affected software components. These boundaries are not explicitly encoded in the code review system but become empirically visible through patterns of interaction, participation, and linkage between code reviews. For each boundary type, we model the structure of the code review environment using graph-based and set-based representations and define measurements to quantify the extent of information diffusion. These representations allow us to apply formal procedures that are independent of platform-specific implementation details.

Table 5.1: Mapping of theoretical constructs to their operationalizations and measurement procedures.

Theoretical Construct	Operationalization	Measurement Procedures
Information diffusion across social boundaries	Dissimilarity of participants between code reviews	Jaccard distance between participant sets
Information diffusion across organizational boundaries	Team affiliation structure within a code review	Number of connected components in team affiliation graph per code review
Information diffusion across architectural boundaries	Repository boundaries between linked code reviews	Proportion of linked review pairs spanning different repositories

We operationalize these constructs using representations based on sets and graphs. For each boundary type, we define

- a model to capture its structure within the data, and
- a measurement to quantify the extent of information diffusion.

The following subsections describe our modelling and measurement procedures for each type of boundary. These definitions form the empirical foundation for answering our research questions and testing our hypotheses. Table 5.1 summarizes how we operationalize the theoretical constructs defined in our framework and the specific measurements we use to test them.

The following subsubsections describe each operationalization and its measurement procedure in detail.

Information Diffusion Across Social Boundaries

To assess the extent of information diffusion across social boundaries, we analyze how dissimilar the sets of developers are across different code reviews. The underlying assumption is that when reviews are conducted by diverse and changing groups of participants, information is more likely to diffuse broadly through the communication network, as it crosses boundaries between distinct developer groups.

Formally, we represent each code review $r \in \mathcal{R}$, where $\mathcal{R}$ is the set of all reviews, by the set P_r of its human participants. To quantify the dissimilarity between two reviews $r_i, r_j \in \mathcal{R}$, we use the Jaccard simmiliarity between their participant sets. Given two sets A and B, the Jaccard similarity is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}. \tag{5.2}$$

Based on this, we define the information diffusion across social boundaries D_{soc} between two reviews r_i and r_j as the Jaccard similarity of their participant sets:

$$D_{\text{soc}}(r_i, r_j) = J(P_{r_i}, P_{r_j}) = \frac{|P_{r_i} \cap P_{r_j}|}{|P_{r_i} \cup P_{r_j}|}$$

A value of 1 indicates completely disjoint sets of participants in the two code reviews, while a value of 0 indicates that two code reviews share all participants. We compute this similarity for all unordered pairs of distinct code reviews, excluding self-comparisons. The resulting distribution of D_{soc} values forms the basis for assessing information diffusion across social boundaries and provides the empirical basis for testing hypothesis H_1.

Information Diffusion Across Organizational Boundaries

To assess the extent of information diffusion across organizational boundaries, we analyze how many distinct teams are involved in each code review.

At Spotify, developers can be affiliated with multiple teams simultaneously (cf. Section 5.4.1). As a result, simply counting the number of affiliated teams in a code review would overestimate the extent of information diffusion across organizational boundaries, since developers with multiple team memberships inherently bridge those boundaries—independent of any interaction that occurs through code review itself. To more accurately capture the information diffusion occurring exclusively through code review, we eliminate the bridging effects of developers with multiple team affiliations, as these connections reflect organizational overlap independent of the review process.

For each code review r, we construct an undirected graph $G_r = (V_r, E_r)$ to represent the team-level collaboration structure within that review:

- V_r is the set of all teams affiliated with at least one participant in review r, formally defined as
$$V_r = \bigcup_{p \in P_r} \tau(p)$$
where P_r is the set of participants in review r, and $\tau(p)$ is the set of teams to which participant p is affiliated.
- For each participant $p \in P_r$ with multiple team affiliations, we add an edge between every pair of teams in $\tau(p)$. That is, if a participant is affiliated with more than one team, all of their teams are connected to one another in the graph.

This graph captures the organizational structure that arises through team co-participation within the specific context of review r, while avoiding artificial connections due to multi-affiliations that exist independently of the review.

We then compute the number of connected components in G_r, denoted $\kappa(G_r)$, which reflects how fragmented the team participation is in review r. We define the extent of organizational boundary diffusion for review r as:

$$D_{\text{org}}(r) = \kappa(G_r)$$

We compute $D_{\text{org}}(r)$ independently for each code review and summarize the resulting distribution using an empirical cumulative distribution function (ECDF) and descriptive statistics. This operationalization provides the empirical basis for testing hypothesis H_2.

Information Diffusion Across Architectural Boundaries

To assess the extent of information diffusion across architectural boundaries, we analyze whether a code review is linked to at least one other review from a different repository.

At Spotify, each repository typically represents a distinct software component in the organization's modular architecture. A link between two code reviews from different repositories therefore signals information diffusion across architectural boundaries—that is, across otherwise decoupled software components.

We restrict this analysis to code reviews that are explicitly linked to at least one other review. Let $\mathcal{R}' \subseteq \mathcal{R}$ be the set of all code reviews with at least one link to another review. For each review $r \in \mathcal{R}'$, we define

- L_r as the set of reviews linked to r, and
- $\rho(r)$ as the repository to which review r belongs.

We define the architectural boundary diffusion $D_{\text{arch}}(r)$ as a binary variable that indicates whether any of the reviews linked to r belong to a different repository:

$$D_{\text{arch}}(r) = \begin{cases} 1, & \text{if } \exists r' \in L_r \text{ such that } \rho(r') \neq \rho(r) \\ 0, & \text{otherwise} \end{cases}$$

This value captures whether code review r bridges architectural boundaries through its links.

We compute $D_{\text{arch}}(r)$ for each linked code review and summarize the distribution using an empirical cumulative distribution function (ECDF) and descriptive statistics. This operationalization serves as the empirical basis for testing hypothesis H_3.

5.5 Results

This section presents the results of our empirical analysis of information diffusion in code review at Spotify. For each type of boundary (i.e., social, organizational, and architectural), we report the measurement results as introduced in Section 5.4.3. These results form the basis for testing our hypotheses and assessing the theory that code review functions as a communication network.

5.5.1 Information Diffusion Across Social Boundaries

To assess whether information diffuses across social boundaries, we analyze how similar the sets of developers are across different code reviews. High dissimilarity suggests that information flows between distinct developer groups, rather than being confined to isolated clusters.

Figure 5.2 shows the empirical cumulative distribution function (ECDF) of the similarity scores D_{soc} across all pairs of code reviews without self-comparison. We observe that the vast majority of review pairs involve entirely distinct sets of participants. Specifically, over 99.62% of all review pairs have no overlap in code review participants. Only 0.04% exceed a score of 0.5, indicating minimal participant overlap. This suggests that code review activity at Spotify is highly distributed across different social developer groups.

5.5.2 Information Diffusion Across Organizational Boundaries

To assess information diffusion across organizational boundaries, we model each code review as a team affiliation graph, where nodes represent teams and undirected edges connect teams that share at least one developer. To exclude multi-team affiliations of developers (cf. Section 5.4.3), we compute the number of connected components in this graph, which we interpret as the number of distinct, non-overlapping teams participating in the code review. A higher number indicates greater information diffusion across organizational boundaries. For

Figure 5.3 shows the empirical cumulative distribution function of the number of not otherwise associated teams per code review. The vast majority (over 81%) of code reviews involve a single team group, while approximately 18% involve developers from multiple distinct teams. Although very rare, some code reviews span over more than three, up to 22 otherwise disconnected teams.

5.5.3 Information diffusion across architectural boundaries

To assess information diffusion across architectural boundaries, we analyze whether linked code reviews span multiple repositories. Since each repository represents a separate software component in Spotify's modular architecture, links between reviews

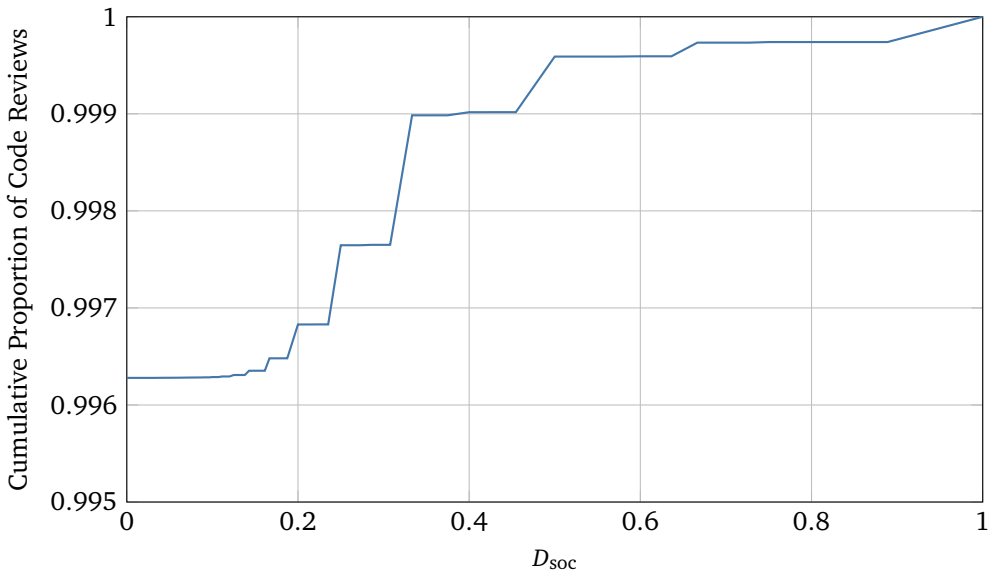


Figure 5.2: Empirical cumulative distribution of information diffusion across social boundaries, based on Jaccard similarity of participants in 220,733 code reviews. over 99.62% of all review pairs have no overlap in code review participants.

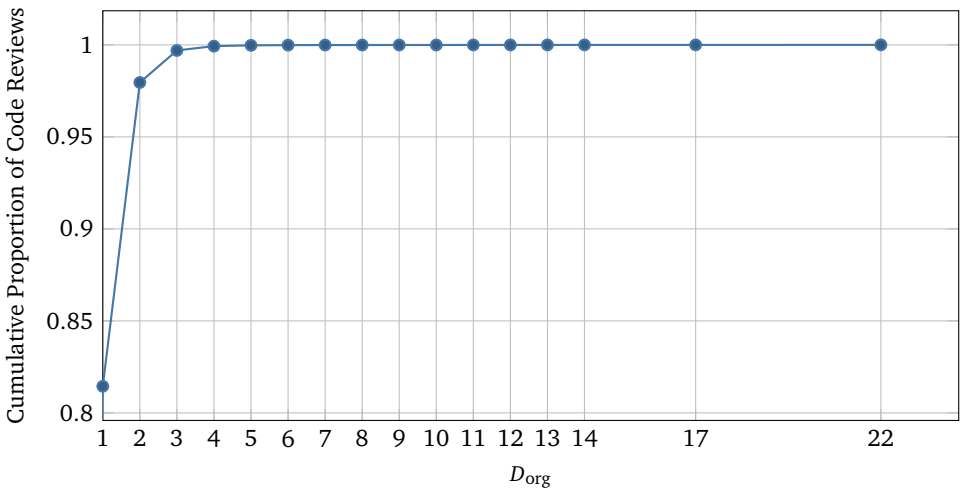


Figure 5.3: Empirical cumulative distribution of distinct teams involved in a code review. Approximately 18% of all code reviews involve developers from more than one team.

from different repositories are interpreted as instances of information diffusion across architectural boundaries.

In our dataset, 5.82% of all code reviews (12,867 out of 220,733) are linked to at least one other code review. Among these, 12,733 links (99.00%) connect reviews from two different repositories. References that link two code reviews from the same repository often originate from few monolithic code repositories at Spotify—large-scale software projects composed of multiple components, such as streaming clients for web, Android, or iOS. Due to confidentiality constraints, however, we are unable to further describe these cases in more detail.

5.6 Discussion

5.6.1 Empirical Evidence for the Theory

This subsection discusses the empirical evidence for supporting the three hypotheses introduced in our study by examining the results of our measurements across social, organizational, and architectural boundaries. Together, these results help assess the extent to which code review functions as a communication network for information diffusion at Spotify.

Hypothesis H_1: Code review enables the diffusion of information across social boundaries between developers.

Our results provide *strong empirical support* for H_1. As shown in Figure 5.2, the overwhelming majority of code review references occur between reviews that involve entirely disjoint sets of participants. Specifically, over 99.62% of all review pairs have no overlap in reviewer identities, and only 0.04% exceed a score of 0.5, indicating minimal participant overlap.

To further contextualize this finding, we examined the number of human participants involved in each individual code review. As shown in Figure 5.4, nearly 25% of all code reviews involve only a single developer (at the time of measurement), and more than 99% involve no more than four. This indicates that most code reviews at Spotify are small in scale, with limited social interaction. While this does not preclude broader information diffusion across Spotify, it suggests that the typical review offers only a narrow channel for such diffusion to occur.

Hypothesis H_2: Code review enables the diffusion of information across organizational boundaries between teams.

Our results provide moderate empirical support for H_2. As shown in Figure 5.3, the majority of code reviews at Spotify involve participants from a single organizational unit. However, approximately 18% of reviews span multiple disconnected team

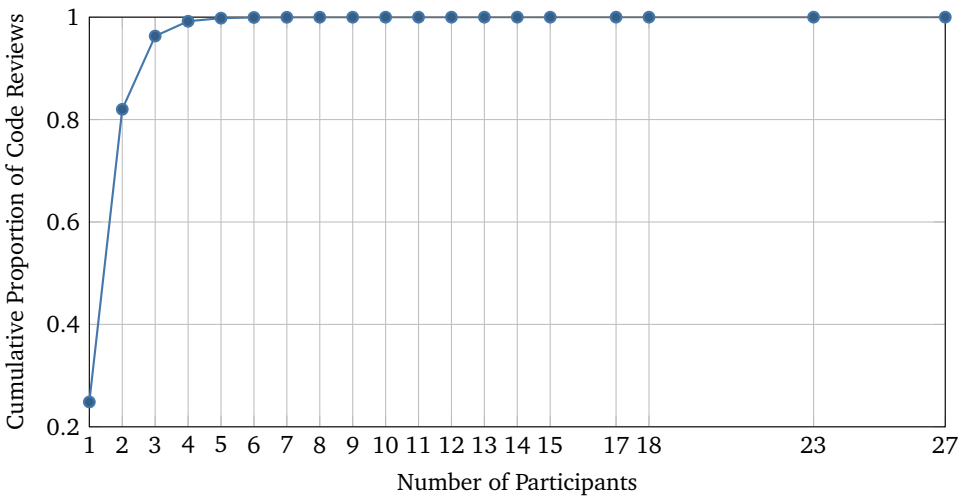


Figure 5.4: Empirical cumulative distribution function of the number of participants per code review.

groups, indicating that information diffuse across formal organizational boundaries. We consider it meaningful given that software components are intentionally designed to exhibit high cohesion and low coupling to minimize the need for cross-team interaction. From this perspective, 18% of reviews involving multiple team groups reflects a nontrivial level of organizational communication during code review. In rare cases, reviews include developers from up to 22 distinct teams, underscoring the capacity of the review process to support wide-reaching coordination. Overall, while code review is primarily a within-team activity, it can serve as an effective cross-team communication channel.

Hypothesis H_3: Code review enables the diffusion of information across architectural boundaries between software components.

We found evidence to support H_3. A total of 99.00% of all linked code reviews connect different repositories, which serve as proxies for distinct software components within the architecture of Spotify's software systems. This high proportion of cross-repository links indicates that, when code reviews are linked, they frequently bridge architectural boundaries. Even the small fraction of intra-repository links primarily originates from large, monolithic codebases composed of multiple tightly integrated components.

However, the linking functionality in the code review platform appears to be rarely used in general, independently of our analysis. Only 5.82% of all code reviews are linked to at least one other code review. This suggests that, while code reviews are technically capable of supporting architectural information diffusion, this function is

not consistently utilized in practice. As a result, the potential for cross-component coordination through code review exists, but remains underexploited.

5.6.2 Alignment with Prior Work

Our findings both corroborate and extend prior research on code review as a communication network. In particular, our results provide quantitative support for the qualitative findings reported by Bacchelli and Bird, who found that practitioners at Microsoft expect code review to serve not only for defect detection but also for sharing information and coordinating across team boundaries. We observed that approximately 18% of code reviews at Spotify involve multiple disconnected team groups—suggesting that such expectations are not only present among developers, but also reflected in actual review participation patterns. This finding demonstrates that cross-team information diffusion through code review does occur in practice, and that these expectations can be corroborated with quantitative evidence in an organizational context beyond Microsoft.

In addition, our analysis aligns with the results of our own previous work [39], in which we modelled code review interactions as a communication network and simulated information diffusion under best-case assumptions. That study revealed a surprisingly large theoretical diffusion potential: under ideal conditions, half of the developers at Spotify could reach between 72% and 85% of all participants within four weeks. At the time, this level of reachability seemed unexpectedly high. However, the current finding of extremely high dissimilarity in participant sets across code reviews helps explain that result. Although individual reviews involve only small numbers of developers, the communication network that emerges across reviews is fine-grained and highly distributed—connecting many otherwise disjoint social groups. This structure enables large-scale diffusion over time, even though each review appears localized in isolation.

Together, these comparisons with prior work reinforce our interpretation of code review as a mechanism that not only supports local collaboration, but also enables meaningful information flow across social, organizational, and architectural boundaries.

5.6.3 Theoretical Implications

The results of this study provide strong empirical support for the theory of code review as a communication network. All three hypotheses derived from the theory—concerning diffusion across social, organizational, and architectural boundaries—were supported by the data. This confirms that code review enables information flow not only within local teams or components, but also across structural boundaries that typically constrain communication in large-scale software development.

The consistency of our findings across all three dimensions indicates that the communication network emerging from code review is both broad and robust. It connects socially disjoint groups of developers, spans across formal team boundaries, and bridges modular software components. Importantly, these connections are not enforced by formal processes or coordination structures, but emerge organically from decentralized, asynchronous review interactions. This supports the theoretical claim that code review is not just a technical gatekeeping mechanism, but an infrastructure for distributed communication embedded in routine development practice.

These results also enhance the explanatory power of the theory. Prior work has emphasized the communicative value of code review in qualitative terms [7, 14, 17, 29, 106]. Our findings demonstrate that this communicative function can be observed, measured, and validated quantitatively at scale. Moreover, the theory appears to generalize beyond its original context: while it has been primarily shaped by studies in smaller or qualitatively analyzed settings, our large-scale, confirmatory evaluation in an industrial environment (i.e., Spotify) confirms that the theoretical expectations hold in practice.

Together, these findings strengthen the theory's validity, suggest it has a broad scope of applicability, and position it as a solid foundation for future work. Going forward, this theory may serve as a basis for studying how communication structures in software engineering emerge from development practices, how they can be supported or shaped by tools, and how they relate to other socio-technical processes such as coordination, decision-making, and knowledge sharing.

5.6.4 Practical Implications

Our findings have several implications for practitioners involved in code review, including developers, team leads, and organizational decision-makers.

First, the evidence that code review facilitates information diffusion across social, organizational, and architectural boundaries reinforces its value beyond defect detection. Developers and teams should recognize code review as a mechanism not only for code quality assurance, but also for communication and coordination across otherwise disconnected parts of the organization. This communicative function is especially important in large-scale, distributed settings where informal communication may be limited. At the same time, it raises concerns about recent trends such as integrating large language models (LLMs) into the review process [30]. While LLMs may increase efficiency, they may also reduce opportunities for human-to-human interaction, potentially undermining the cross-boundary communication that makes code review such a valuable communication medium—a concern that was also recently rised in Heander, Söderberg, and Rydenfält.

Second, organizations may benefit from treating code review as part of their broader communication and coordination infrastructure. The fact that information naturally

flows across boundaries suggests that code review participation and code review assignment policies could be optimized not only for code ownership, but also for maximizing cross-cutting information exposure. For example, involving developers from different teams or software components in strategically selected reviews could support information exchange and architectural consistency.

Third, our findings have implications for legal and compliance functions in multinational enterprises. If code review functions as a communication network, it provides a measurable proxy for developer collaboration—potentially even across international borders. This is legally relevant, because cross-border collaboration within multinational enterprises may trigger, for instance, tax obligations as profits can become taxable across jurisdictions [37]. Given the financial risks associated with tax non-compliance—as exemplified by the court case from 2023 in which the U.S. Internal Revenue Service (IRS) claims that Microsoft owes an additional $28.9 billion in taxes, along with penalties and interest [124]—organizations need robust, theory-based methods to identify such cross-border collaboration. The idea of using cross-border code reviews (i.e., code reviews with participants employed by different subsidaries in different countries) as a proxy for international collaboration was previously proposed by Dorner et al. With our findings, this proposal is now grounded in a more solid theoretical foundation, supported by empirical evidence that code review enables cross-boundary information diffusion across organizational and potentially jurisdictional borders.

5.7 Threats to Validity

As with any empirical study, our work is subject to several threats to validity. In the following, we discuss these threats to construct, internal, external, and conclusion validity.

5.7.1 Construct Validity

Construct validity concerns whether the study accurately captures the concepts it intends to measure. In our study, several modelling assumptions may affect construct validity.

First, we acknowledge that there is no universally accepted definition of what constitutes a “code component” in software engineering, as, for example, pointed out by Broy et al. The term can refer to different granularities and technical aspects such as classes, modules, packages, services, or repositories. In our study, we treat each repository as a proxy for a code component. While this operationalization is pragmatic and consistent among different technologies, it may not capture all relevant structural or architectural boundaries, potentially affecting the comparability of our findings.

Second, our unit of analysis is the team, the lowest ancestor in the organizational hierarchy. This abstraction supports aggregation and analysis of developer behavior at a manageable level, but may overlook relevant dynamics occurring at higher organizational levels (e.g., departments).

Third, we rely on GitHub's automatic linking infrastructure to identify references between code reviews.[5] We assume this system is a reliable source for capturing developer-intended links, as it is widely used in practice and maintained as part of GitHub's platform functionality. This assumption is also supported by prior research [68, 136], although it is important to note that these studies focused on @-mentions, which link to users, rather than references to pull requests or issues as we do.

5.7.2 Internal Validity

Internal validity refers to the extent to which causal relationships can be attributed to the observed phenomena rather than to confounding factors. In our study, team affiliation data are derived from an internal system that was partially and manually sanity-checked. While we took care to validate the mappings, we ultimately rely on infrastructure that cannot be externally audited, introducing the possibility of assignment errors.

Furthermore, some automated activity may have been carried out through human accounts. In such cases, it becomes difficult to distinguish automation from regular developer contributions, which could impact the accuracy of our observations. In Dorner et al., for instance, we identified 14 accounts at Microsoft that submitted more than 500 code reviews during a four-week observation window, averaging over three reviews per hour. This level of activity strongly suggests some form of automation. However, in the current study at Spotify, we found no evidence of similar behavior, and no accounts exhibited such extreme review volumes. While this reduces the likelihood of automation-related distortion in our dataset, we cannot rule it out entirely.

5.7.3 External Validity

External validity addresses the generalizability of our findings beyond the context of the study. In alignment with Karl Popper's philosophy of science, we explicitly refrain from making general claims. According to Popper, scientific knowledge progresses through conjectures and refutations, not through inductive generalization. Our goal is thus to offer falsifiable propositions or hypotheses grounded in empirical observations. We welcome future studies to attempt their replication or refutation in other settings, organizations, or technological stacks.

[5] https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/autolinked-references-and-urls#issues-and-pull-requests

5.7.4 Conclusion Validity

Conclusion validity pertains to the strength and credibility of the inferences drawn from the analysis. We do not perform formal statistical hypothesis testing in this study. Instead, we rely on qualitative hypothesis evaluation, supported by detailed descriptions of our reasoning and by making all relevant data and steps transparent. While this approach supports interpretability, it also leaves some room for subjective interpretation and researcher bias.

5.8 Conclusion

In this paper, we formalized and empirically tested the theory of code review as a communication network. Drawing from a large-scale dataset of code review at Spotify, we examined the extent to which code review supports information diffusion across social, organizational, and architectural boundaries. These boundaries represent common barriers to collaboration in large-scale software development, and their traversal is a necessary condition for any communication network to facilitate broad information flow.

Our results provide substantial support for the theory of code review as a communication network. We found that code review connects largely disjoint sets of participants, indicating diffusion across social boundaries. A meaningful portion of code reviews also involve developers from different teams, suggesting that information can flow across organizational boundaries. Finally, nearly all linked code reviews span different repositories, which serve as proxies for software components—confirming the capability of code review to traverse architectural boundaries. Taken together, these findings show that code review enables information diffusion across multiple types of boundaries and thus exhibits the characteristics of a communication network.

Future work can extend this research in several directions. One important avenue is to study the content of information being diffused—e.g., whether it pertains to design rationale, architectural decisions, or coordination signals. Another is to investigate under what conditions diffusion is most effective, and how it interacts with organizational structure, tooling, and developer incentives. Finally, generalizing the findings across different platforms, domains, and organizational settings would help assess the theory's scope and identify potential boundary conditions.

By formalizing and empirically evaluating the theory of code review as a communication network, we hope to contribute to a more robust theoretical foundation for understanding the role of code review for collaborative software engineering and to motivate further confirmatory research in empirical software engineering.

Data Availability

The replication package, including code, datasets, and analysis scripts, is publicly available at

`https://github.com/michaeldorner/measuring-information-diffusion-in-code-review-at-spotify`.

Please note that the complete replication package will be archived on Zenodo following the completion of the review process.

Acknowledgments

We are grateful to Spotify for their support and for providing the data that made this study possible, and to the anonymous reviewers at ESEM 2025 of the registered report for their constructive feedback, which greatly improved the paper.

This work was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

Code Review as Proxy for Collaborative Software Engineering

6

This chapter is based on Michael Dorner, Maximilian Capraro, Oliver Treidler, Tom-Eric Kunz, Darja Šmite, Ehsan Zabardast, Daniel Mendez, and Krzysztof Wnuk. *"Taxing Collaborative Software Engineering"*. In: *IEEE Software* (2024), pp. 1–8. DOI: 10.1109/MS.2023.3346646.

Abstract

The engineering of complex software systems is often the result of a highly collaborative effort. However, collaboration within a multinational enterprise has an overlooked legal implication when developers collaborate across national borders: It is taxable. In this article, we discuss the unsolved problem of taxing collaborative software engineering across borders. We (1) introduce the reader to the basic principle of international taxation, (2) identify three main challenges for taxing collaborative software engineering making it a software engineering problem, and (3) estimate the industrial significance of cross-border collaboration in modern software engineering by measuring cross-border code reviews at a multinational software company.

6.1 Introduction

> "He's spending a year dead for tax reasons."
>
> – Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Modern software systems are often too large, too complex, and evolving too fast for single developers to oversee. Therefore, software engineering has become highly collaborative. Further, software development is often a joint effort of individuals and teams collaborating across borders, especially in multinational companies with their subsidiaries spread around the globe [61]. However, collaboration has a legal implication if individuals collaborate across borders: the profits from those cross-border collaborations become taxable.

In this article, we describe the complexity of applying the established international taxation standards required and enforced by national tax authorities in the context of modern software engineering with its distributed and fine-grained collaboration crossing borders. We start with a gentle introduction to international standards in multinational taxation and its basic *arm's length principle* for software engineers. We then discuss the challenges of taxing collaborative software engineering and illustrate the industrial significance of cross-border collaboration in an industrial case, namely code review.

Taxation in software industry has been debated for many decades [86]. The problem with taxing the final result of software engineering, the software product or service, for example, has shown to be challenging to tackle and is still subject to ongoing and broad discussion [88]. Here, we extend the debate to software engineering, the way the software products and services are being developed, which has not yet been covered. Our goal is to raise a debate and draw attention to this problem among a software engineering audience. For in-depth information on basic transfer pricing concepts including standard methods and tax compliance requirements, we recommend the interested reader further readings [123].

6.2 A Gentle Introduction to Taxing Multinational Enterprise for Software Engineers

Consider *devnullsoft Group*, a multinational enterprise that develops and sells a software-intensive product, has two legal entities: *devnullsoft GmbH* in Germany and its subsidiary *devnullsoft AB* in Sweden. The German development team employed by *devnullsoft GmbH* develops the software-intensive product jointly with the Swedish development team employed by the Swedish subsidiary *devnullsoft AB*. The German *devnullsoft GmbH* sells this resulting product to customers.

Without any further consideration, solely the German *devnullsoft GmbH* generates profits, which are then fully taxed in Germany according to German law. The Swedish tax authorities are left out in the cold because *devnullsoft AB* has no share of the profit that could be taxed in Sweden, although *devnullsoft AB* contributed significantly to the product through code contributions, code reviews, bug reports, tests, architectural decisions, or other contributions that made the success of the software possible

To avoid this scenario and to provide a common ground for international taxation, reducing uncertainty for multinational enterprises, and preventing tax avoidance through profit shifting, nearly all countries in the world agreed on and implemented the so-called *arm's length principle* as defined in the *OECD Transfer Pricing Guidelines for Multinational Enterprises and Tax Administrations* [87].

The *arm's length principle* is the guiding principle and the de-facto standard for the taxation of multinational enterprises that requires associated enterprises to operate as if not associated and regular participants in the market from a taxation perspective. This principle ensures that transfer prices between associated companies of multinational enterprises are established on a market value basis and not misused for profit shifts from high to low tax regions.

To comply with the arm's length principle, *devnullsoft GmbH* in Germany and *devnullsoft AB* in Sweden need to operate from a taxation perspective as if they were not associated. Since a regular participant in the market would not provide code contributions, code reviews, tests, or architectural designs free or other contributions of charge to a closed-source software project, *devnullsoft GmbH* in Germany needs to pay for the received contributions, the so-called *transfer price*.

Transfer prices are the prices at which an enterprise transfers physical goods and intangibles or provides services to associated enterprises. Since software is intangible itself, the transfer of intangibles like source code, code reviews, bug reports, etc., is our focal point. This transfer price guarantees that *devnullsoft AB* gets its share of the profit, which then can be taxed by the Swedish tax authorities.

In Figure 6.1, we provide a schematic overview of transfer pricing between the two associated software companies from our example. Although *devnullsoft AB* contributed

significantly to the software-intensive product, without a transfer price, *devnullsoft AB* has no share of profits; all profits are fully taxable in Germany only. However, if *devnullsoft GmbH* in Germany pays a transfer price reflecting the value for the services and intangible properties received from its Swedish associated enterprise, *devnullsoft AB* realizes profits which then are taxable in Sweden.

In our case, the *devnullsoft Group* does not artificially shift profits to a tax haven. Yet, one can easily imagine that neglecting to charge arm's length prices can be intentionally misused for profit-shifting. Therefore, the OECD guidelines permit tax authorities like the Swedish tax authority to adjust the transfer price where the prices charged are outside an arm's length range. Such an adjustment will carry interest and might be coupled with penalties. In the wake of the OECD's *Base Erosion and Profit Shifting (BEPS) Project*[1], the regulatory framework has become considerably stricter at an international and national level. As a result, tax authorities can demand more comprehensive information to detect misalignments and enforce tax adjustments. From the companies' perspective, its software development may be—intentionally or unintentionally—noncompliant and face the risk of being legally prosecuted.

6.3 Challenges

So, what are transfer prices for collaborative software engineering that comply with this arm's length principle? Determining a market price for intangibles is inherently difficult and is reflected in a broad price range. Collaborative software engineering, however, scales the problem of a transfer-price determination to a new level of complexity because the reality of modern software engineering is significantly more complex than our introductory example above may suggest. Since transfer price regulations apply to a much broader definition of intangibles compared to accounting standards, the latter can not be used as a reliable measure of value for transfer pricing purposes [87].

In the following, we discuss three main high-level challenges for transfer pricing in collaborative software engineering within multinational enterprises. Figure 6.2 highlights the complexity in modern collaborative software engineering at *devnullsoft Group* and where those three challenges apply.

Challenge 1: What is a taxable transaction in software engineering?

The trouble for transfer pricing in software engineering begins with a fundamental question: What is actually a taxable transaction in the context of collaborative software engineering? We simply do not know what types or characteristics types or characteristics classify a taxable exchange of intangibles or services across the boundaries of a country in the context of software engineering.

[1] `https://www.oecd.org/tax/beps/`

Figure 6.1: A schematic overview of the necessity and mechanics of transfer pricing in a multinational enterprise (*devnullsoft Group*) with two associated software companies (*devnullsoft AB* in Sweden and *devnullsoft GmbH* in Germany): Without considering a market-based compensation, the so-called *transfer price*, *devnullsoft AB* has no share on the profits which could be taxed by the Swedish tax authorities; all profits are with *devnullsoft GmbH* and, therefore, all taxes stay within Germany.

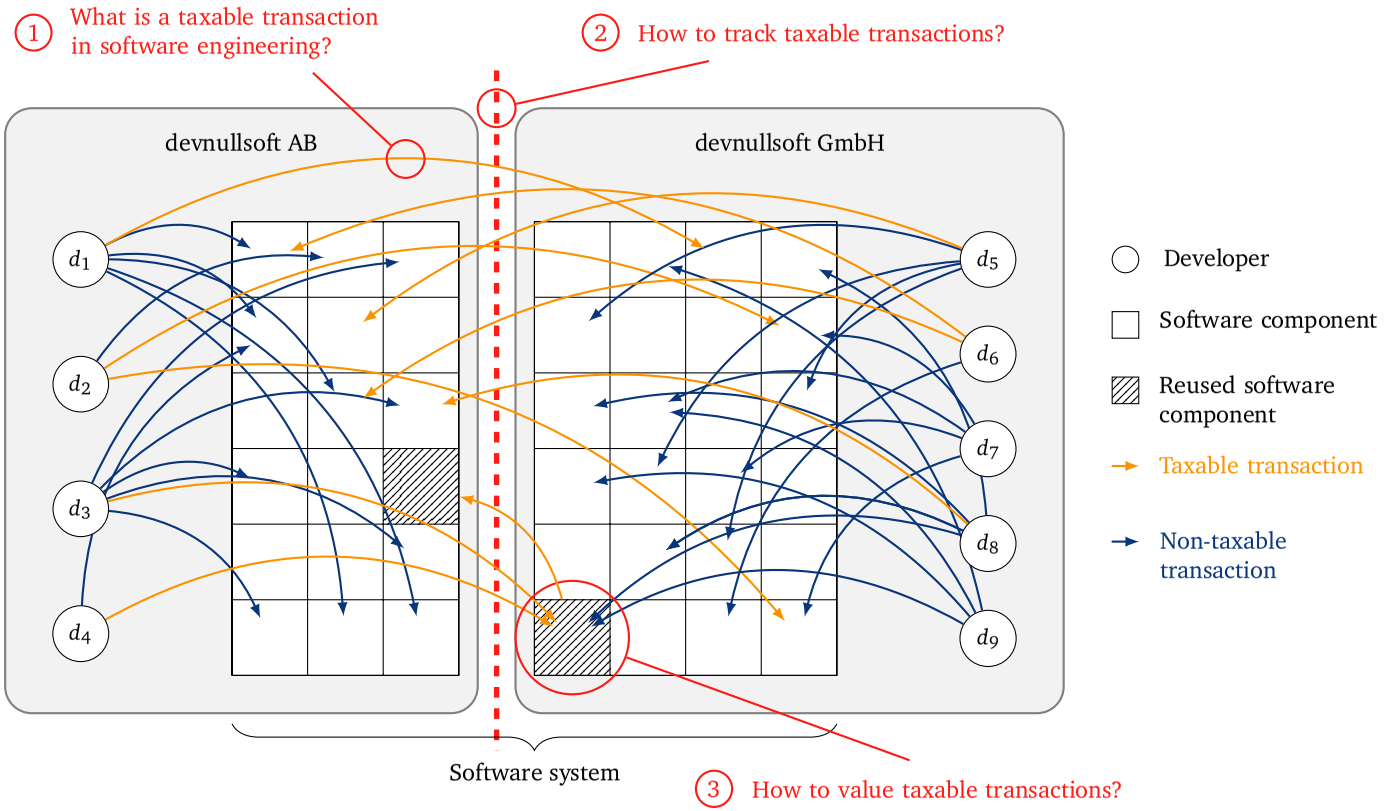


Figure 6.2: A schematic overview of collaborative software engineering and three challenges for transfer pricing specific to software engineering.

Among other potentially relevant types of taxable intangibles, such as *goodwill* or *group synergies*, we discuss in this and the following subsections two types of intangibles that are highly relevant for software engineering: know-how and licenses.

The OECD Transfer Pricing Guidelines define *know-how* as the "proprietary information or knowledge that assist[s] or improve[s] a commercial activity, but that [is] not registered for protection in the manner of a patent or trademark". The commercial activity includes the manufacturing, marketing, research and development of and for a software system.

Does the OECD definition imply that all types of information exchanged during collaborative software engineering are know-how? On the one hand, yes, since all information is proprietary and, to some extent, contributes to the software being developed or its engineering processes. But on the other hand, how do we know which information assists or improves the commercial activity, meaning the engineering of the software system, over time? For example, a quick and dirty bug fix without sufficient documentation or testing may improve the software system in the present but makes changes more costly or even impossible in the future. Making such sub-optimal decisions leads to incurring technical debt [102], which is potentially relevant for taxation. Thinking the concept of technical debt ahead, we have begun to understand that similar to physical, tangible assets, software assets degrade and lose value inevitably due to intentional or unintentional decisions caused by technical or non-technical manipulation of the asset or associated assets during all stages of the product life-cycle [133] Such an asset degradation will also be of great interest from a taxation perspective.

The second type of intangibles highly relevant to transfer pricing in software engineering if transferred across borders is *licenses*. Although maybe not even explicit, the company-internal use and reuse of components is an instance of licensing. Complex software systems are not monolithic blocks of code but consist of components that are developed, shared, and reused by separate teams. However, we lack a common understanding of software components and reuse in software engineering for taxation. Not every component is directly used for or in a software-intensive product, but maybe adds value to the product. For example, a well-engineered CI/CD pipeline accelerates the development cycles and brings new features or bug fixes faster to the customers [47]. Furthermore, it is also not always clear who owns, contributes to, or uses a component within a company, and the roles may even change over time [134]. In contrast to open source, the reuse is often implicit, lacking a company-wide license agreement that clarifies the responsibility and accountability between component owners and users. Even worse, we do not even know if our definition and understanding of code ownership [85] suffices the definition of ownership in a taxation context.

Additionally, we see an interplay between those two types of intangibles, know-how and licensing, since they may be two sides of the same collaboration. For example, when code contributions from the component user support instance of reuse.

Insight

Identifying the taxable transactions requires either a holistic perspective of software engineering or at least suitable, practical, and accurate proxies. Compliant software engineering needs a common understanding and a taxonomy of taxable transactions specific to software engineering.

Challenge 2: How to track cross-border transactions in software engineering?

Also, the practical tracking of taxable know-how and licensing (and potentially other types of intangibles) is a challenge on its own.

Tracking know-how is an inherently difficult task. Since the teams collaborating are no longer colocated, numerous tools enable an exchange of know-how in software engineering. Those tools are suitable as rich data sources to different extents: While domain-specific tools like issue trackers or collaborative software development platforms like GitHub or Gitlab often track the exchanges very thoroughly, other communication and collaboration tools do not: Online meetings, for example, can facilitate an exchange of taxable intangibles, but this exchange is not tracked by any tool. But even if there is a rich data basis available, leveraging those data sources is problematic for following reasons:

- *Establishing location*—It can be difficult to establish the location of collaborators or capture when a location of a collaborator has changed, because organizations often preserve only the latest version of the organizational structures.

- *Privacy*—Analyzing the complete communication of developers may be perceived as a measure of surveillance, which raises ethical and legal concerns related to privacy.

In contrast to the potentially rich sources for tracing taxable transactions from collaboration tools used in software engineering, tracking company-internal reuse often lacks a solid data basis. Although companies often track the reuse of external open-source components for open-source license compliance purposes, those tools are rarely used or suitable for tracking company-internal reuse. Also, only reuse that crosses borders is taxable; information that is often not available or stored over time—although component ownership is not static and may be subject to change.

Insight

Data for tracking taxable transactions may be incomplete, faulty with respect to location, or restricted. There is no dedicated tool support yet for the practical transfer price determination.

Challenge 3: How to value taxable transactions in software engineering?

While it is inherently difficult to tax intangibles in general, things are even more complicated in software engineering. Potentially taxable intangibles cover a large range of granularity in software engineering: They may be as large as a microservice providing user authentication used by microservices of other teams (→ intangible *licenses*) or as small as a code change, code review, or bug report (→ intangible *know-how*). Although the code change or feedback in a code review is small—maybe even only one line of code, like in the case of the *Heartbleed* security bug in the OpenSSL cryptography library from 2014 [22]—the potential impact on the software system can be tremendous or even fatal. A software change or a code review delivers value through impact, not size.

The same applies to licensing. The number of use relations of a software component or its size (however defined) does not reflect the value provided to the software-intensive product. While the software component for user authentication may be important for operating the software-intensive product and, therefore, has a large amount of depedent software components, it is not differentiating and may even be considered a commodity.

This means we cannot simply use purely quantitative measurements for transfer pricing. However, the sheer mass of small, fine-grained transactions of all types makes a human qualitative case-by-case evaluation impossible.

Insight

A purely quantitative valuing can hardly reflect the value of transactions; however, a purely qualitative assessment does not scale with the magnitude of cross-border transactions in modern software development.

6.4 An Industrial Example of Cross-Border Collaboration

So, is cross-border collaboration and, therefore, also the taxation of it, a real issue? To estimate the prevalence of cross-border collaboration, we measure cross-border code reviews as proxy for cross-border collaboration in a typical industrial setting.

A cross-border code review is code review with participants from more than one country. Although it originated in collocated, waterfall-like code inspections, its

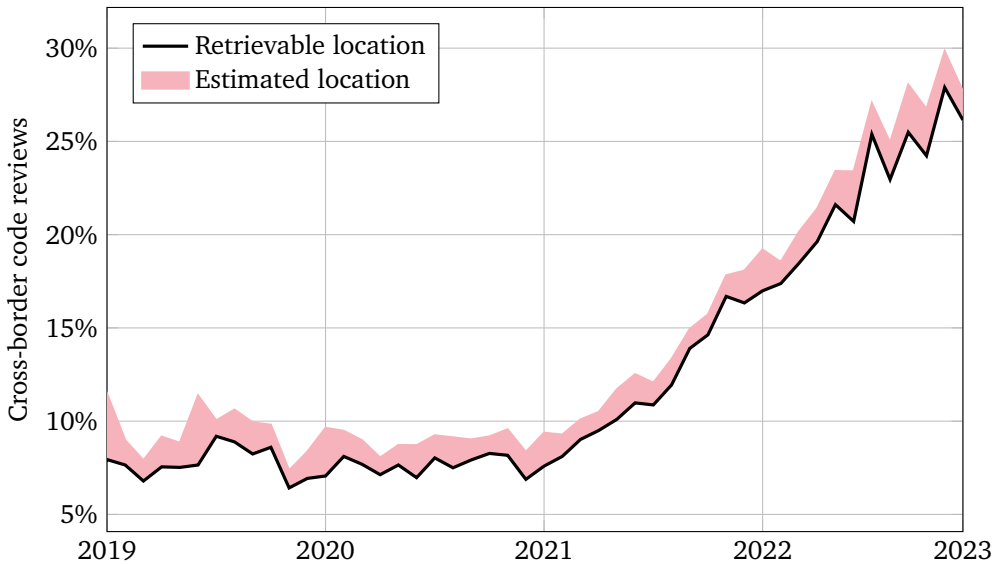


Figure 6.3: The share of cross-border code review at our case company in the years 2019, 2020, 2021 and 2022 (black line) monthly sampled. Since not all historical locations of all code review participants could be reliably retrieved, the share of cross-border reviews could be more significant (indicated by the red area).

modern stances are lightweight and asynchronous discussions among developers around a code change. Different tools are in use, for example, Gerrit or Github and Gitlab with their implementation of code review as so-called pull and merge requests, respectively. Although code review is by far not the only type of collaboration that may include taxable transactions and is also likely not sufficient to determine company-wide transfer prices, the following characteristics make code review a suitable first proxy for cross-border collaboration:

- More than code only—Code review not only includes the actual code change and its authors but also includes the feedback from reviewers that may have formed or changed the code change significantly but is no longer visible in the repository after merging the code change into the codebase. Therefore, our proxy goes beyond existing code-based measurements for collaboration, such as [20].

- Accessible & complete—The code review discussions are (company-internally) public by default and are, thus, accessible. Unlike other tools like instant messaging services or e-mail, code review does not split into public and private, whose analysis may cause privacy concerns.

- Persistent—Code review tools are the backbone of modern code review and ease data extraction. Other types of code review (for example, private or synchronous discussion around a code change through meetings or instant messaging) may not be captured through the tooling, though.

We measured the share of cross-border code reviews at a multinational company delivering software and related services worldwide with main R&D locations in three countries. For many years the company has tried to allocate products to particular sites to avoid the burden of cross-border collaboration. However, our analysis shows that developers represent more than 25 locations because the new corporate work flexibility policy permits relocations [111]. The company uses a single, central, and company-wide tool for its internal software development and code review. Understandably, our case company wants to remain anonymous. Therefore, we are not able to describe the case any further. However, we believe that our case company is exemplary for a multinational enterprise developing software.

From the code review tool, we extracted all code reviews that were completed in 2019, 2020, 2021, and 2022 including their activities. All bot activities were removed and were not considered in our analysis. We then modelled code reviews as communication channels among code review participants [41]. We consider a code review as a discussion thread that is completed as soon as no more information regarding a particular code change is exchanged (i.e., the code review is closed). We complement each code review participant with the information of the country of the employing subsidiaries at the time of the code review.

We provide a replication package to reproduce our results for any GitHub Enterprise instance[2]. Due to the sensitive topic, we are not able to share our data.

Figure 6.3 shows an increase in relative cross-border code reviews over time. The share of cross-border code reviews was between 6% and 10% in 2019 and 2020. Yet, we see a further steep increase reaching between 25% and 30% at the end of 2022.

Interestingly, 6% of all cross-border code reviews involve participants from more than two countries. This means transfer pricing in collaborative software engineering becomes not only a bilateral but a multilateral problem with not only two but multiple—in our case company up to six—different jurisdictions and tax authorities involved in the transfer pricing process.

Although the share of cross-border collaboration may vary among companies, yet, our findings suggest that—through the proxy of cross-border code reviews—cross-border collaboration becomes a significant part of daily life in multinational software companies. It is fair to assume that a further increase in cross-border collaborations in software engineering will draw the attention of tax authorities.

6.5 Conclusion

On the one hand, the arm's length principle is the de-facto standard for multinational enterprises that any multinational company must comply with. On the other hand, software engineering is highly collaborative—beyond geographical and organizational boundaries. Determining a reasonable transfer price for this cross-border collaboration brings the general challenge of taxing intangibles to a new level of complexity.

Pretending to be dead for tax reasons is no option because ignoring the significant cross-border collaboration in modern software development, as we exemplarily found, is a slippery slope: Cross-border collaborations in software engineering will draw the attention of tax authorities. Also, ceasing or forbidding all cross-border collaboration in software engineering is not a valid solution: Reversing the collaborative nature of modern software engineering is likely too costly and takes too long.

Obviously, there are neither simple solutions for such a complex and interdisciplinary problem, nor a single article that can solve this complex problem potentially affecting every software-developing company with developers employed by subsidiaries in more than one country. However, our article aims to bring this eminent and unsolved problem of taxing collaborative software engineering to the audience that can solve this issue. As a software engineering community, we will need to find a common understanding of what constitutes taxable transactions and each company that develops software collaboratively within more than one country needs to learn how to track

[2] See `https://github.com/michaeldorner/tax_se`

and value cross-border collaboration, how to estimate the transfer pricing, and how to report these to the tax authorities to be compliant.

Acknowledgments

We thank our industry partner for providing the data for this study and interpreting the results. We also thank the anonymous reviewers for their fruitful feedback. This work was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

6

Quo Vadis, Code Review?

7

This chapter is based on Michael Dorner, Andreas Bauer, Darja Šmite, Lukas Thode, Daniel Mendez, Ricardo Britto, Stephan Lukasczyk, Ehsan Zabardast, and Michael Kormann. "*Quo Vadis, Code Review?*" Manuscript under review at *IEEE Software*. 2025.

Abstract

Code review has long been a core practice in collaborative software engineering. In this research, we explore how practitioners reflect on code review today and what changes they anticipate in the near future. We then discuss the potential long-term risks of these anticipated changes for the evolution of code review and its role in collaborative software engineering.

7.1 Introduction

> "Nothing is constant except change."
>
> —Heraclitus

During the early days of software engineering, code was often designed and implemented by individual developers working in isolation. As the software systems became more and more complex, software engineering became a collaborative effort requiring many developers with diverse skill sets and specializations, organized in different teams, and later even distributed around the globe. To maintain and enhance a collective understanding of the codebase and the changes to it, developers need to discuss a code change before it gets merged into the codebase. Over the years, these discussions have taken various forms ranging from the formal and heavyweight *code inspections* in the 1980s [45] to synchronous and steady exchanges between two developers in *pair programming* [127]. Nowadays, the asynchronous and less formal discussions around code changes are known as *code reviews*, a practice that has seen widespread adoption in collaborative software engineering [13, 39].

However, change is constant in software engineering—and code review is no exception. Advances in large language models (LLMs) and new regulatory frameworks, such as the EU Cyber Resilience Act, are beginning to influence how code reviews are conducted.

In this article, we take a glimpse into the future of code review. To that end, we begin by analyzing how practitioners reflect on code review today and what they expect to change, based on a survey of 92 professional software developers from four large software-driven companies, each representing a distinct organizational and software engineering context. Building on this analysis, we discuss potential long-term risks for the future role of code review as a core practice in collaborative software engineering.

7.2 Study Design

To understand how developers experience code review today and how they expect it to change, we surved a sample of 92 practitioners.

Table 7.1: Organizational and Software Engineering Contexts and Participant Numbers by Company

Attribute	**SAP**	**Ericsson**	***A Bank***	**JetBrains**
Industry Domain	Enterprise Software	Telecommunications	Banking and Finance	Developer Tools
Business Model	Software products (B2B SaaS, ERP)	Infrastructure provider (network equipment, telecom software)	Financial services (retail and investment banking)	Commercial software vendor (IDEs and tooling)
Primary Software Focus	Customer-facing enterprise platforms	Embedded and network control software	Internal financial transaction and risk management systems	Developer productivity and language tooling
Number of Employees	>100,000	>100,000	>25,000	>2,200
Number of participants	32	24	25	11

To gain a diverse perspective on the future of code review grounded in real-world practice, we employed *quota sampling*, a non-random, two-stage strategy. First, we purposively selected four relevant companies representing varied organizational and software engineering contexts. Then, we aimed to recruit approximately 25 professional developers from each company, closing recruitment after eight weeks or once the quota was reached.

Our sample includes 92 developers from (1) one of the world's largest enterprise software providers (SAP), (2) a globally leading provider of telecommunications infrastructure (Ericsson), (3) one of Europe's largest banks, which has requested anonymity, and (4) a mid-sized vendor of professional developer tools, including widely used IDEs (JetBrains). The number of participants per company varied slightly. See Table 7.1 for an overview of the participating organizations, their contexts, and participant numbers. Though not statistically representative and with varying participation per company, the sample captures diverse industry perspectives on code review.

To capture developers' current code review practices and their expectations for how the practice may evolve over the next five years, we asked six questions:

1. How many hours on average do you currently review code per week? (numerical response)
2. Do you expect to spend more, less, or the same amount of time on code review in five years (compared to today)? (Multiple choice)
3. What software artifacts do you review today? (Multiple selection)
4. What software artifacts do you anticipate reviewing in five years? (Multiple selection)
5. What major changes do you anticipate in code review in five years? (Open-ended)
6. Based on those changes, what implications for code review do you see? (Open-ended)

The survey covered six types of software artifacts covered in code review: Production code, Test code, Parameter and configuration files, Documentation, and GUI-based test code (end-to-end testing). Respondents could also add further artifact types or choose to select none.

The survey was conducted end of 2024 and beginning of 2025 via an online questionnaire, shared through internal communication channels at the respective companies. Participation was both voluntary and anonymous, allowing respondents to share their perspectives freely. To further protect confidentiality, we deliberately excluded demographic questions such as age, gender, or company affiliation, and report the

results on sample level rather than by individual company. Participants were informed about the study's purpose and assured that their participation was voluntary, and their responses would remain anonymous.

Quantitative and multiple-choice responses were analyzed using descriptive statistics. Open-ended responses were examined using *thematic analysis* [26] to uncover common patterns and emerging trends.

All anonymized data and analysis scripts are publicly available at

`https://github.com/michaeldorner/quo-vadis-code-review`.

7.3 Code Review Five Years From Now

What do developers actually do in code review today and how do they think that will change? In this section, we report the results of our survey, capturing how developers currently experience code review today and how they expect it to change in the coming years. See Section 7.2 for more details on study design. The analysis is structured around three central questions from our survey: the *time* they currently spend on code review and how they expect that to change; the *types of artifacts* they review today versus those they anticipate reviewing in the future; and *who* they expect to be involved in the code review process going forward.

7.3.1 Not Less Time

Currently, respondents report spending a median of approximately three hours per week on code reviews. To contextualize this finding, we compare it against two external data sources: first, a 2013 survey by Bosu et al. involving 416 Microsoft developers; and second, the 2019 Stack Overflow Developer Survey, which includes responses from 49 790 developers[1]. Figure 7.1 shows the cumulative distributions of weekly review hours across all three data sources. The distribution in our sample is consistently lower compared to both the 2013 Microsoft study by Bosu et al. and the 2019 Stack Overflow Developer Survey, which show strikingly similar distributions with a median of around four hours spent on code reviews. The reasons for this difference are not evident from our data. Possible explanations may include a decline in the perceived importance of code review over the past years, differences in domain-specific practices, or inaccuracies in self-reported estimates.

A vast majority of practitioners do not expect to spend less time on code review in the coming years. According to our survey:

- 47 % expect to spend *more time*,
- 31 % expect to spend *about the same amount* of time, and

[1] `https://survey.stackoverflow.co/2019`

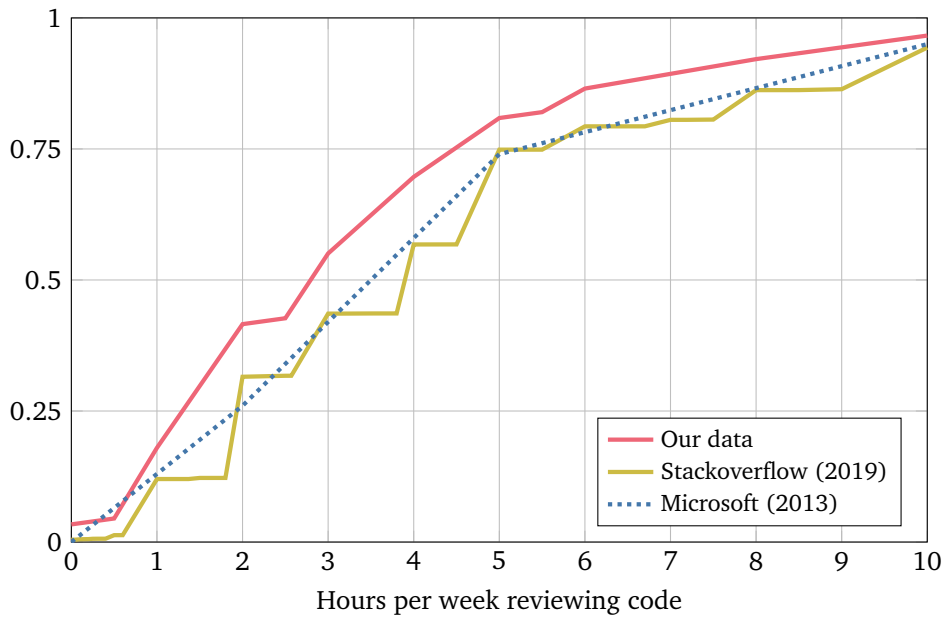


Figure 7.1: Cumulative distribution of hours spent on code review today (≤ 10 hours). In the absence of raw data, we approximated the cumulative distribution from the reported ordinal-scale results at Microsoft [17].

- 22 % foresee spending *less time*.

Although those results suggest that code review might play even a more central role in collaborative software engineering in the near future, we are cautious about extrapolating this insight to the entire software industry. The distributions in our sample are consistently lower than those in our reference datasets—though these datasets are older—which raises the possibility that the companies in our study are simply catching up to broader industry standards rather than indicating a widespread shift in code review practices.

7.3.2 Broader Coverage

Practitioners expect to review a broader range of artifacts in the near future in comparison to today. Over the next five years, we see an increasing interest in reviewing more artifact types, including production code, test code, configuration files, documentation, and GUI test code. Notably, the share of practitioners who currently do not review any artifacts (around 6%) is expected to decrease.

Production code will remain the primary artifact under review in the future. There is also a broad consensus that parameter and configuration files, documentation, and test code will continue to be reviewed regularly. A particularly notable trend is the comparatively sharp increase in the future prediction of review focus on GUI-based test code, suggesting a growing recognition and integration of it in future software development practices.

The connected dot graph in Figure 7.2 highlights the expected changes in the coverage of reviewed artifacts in the future.

7.3.3 The New Code Review Participant

Given the current hype around generative AI, it is hardly surprising that neary all practitioners, when asked about anticipated major changes in future code reviews, expect large language models to become active participants in the process. This trend is also highlighted by Davila, Melegati, and Wiese [30], whose gray literature review identified strong practitioner interest in both proposing novel generative AI–based solutions and adopting established tools such as ChatGPT for code review.

In contrast to the deterministic, rule-based, or pattern-matching bots that long assisted in code reviews [8], survey respondents envision LLMs taking a way more active role: An LLM might act as the code reviewer, assessing the code written by humans and providing feedback through an *automated review*. The anticipated autonomy of LLMs varies and can be viewed along a spectrum, ranging from supporting human reviewers as an additional assistant (“*AI might take over some parts of the code review process.*”) to fully conducting the review independently (“*complete code review with the help of AI*”).

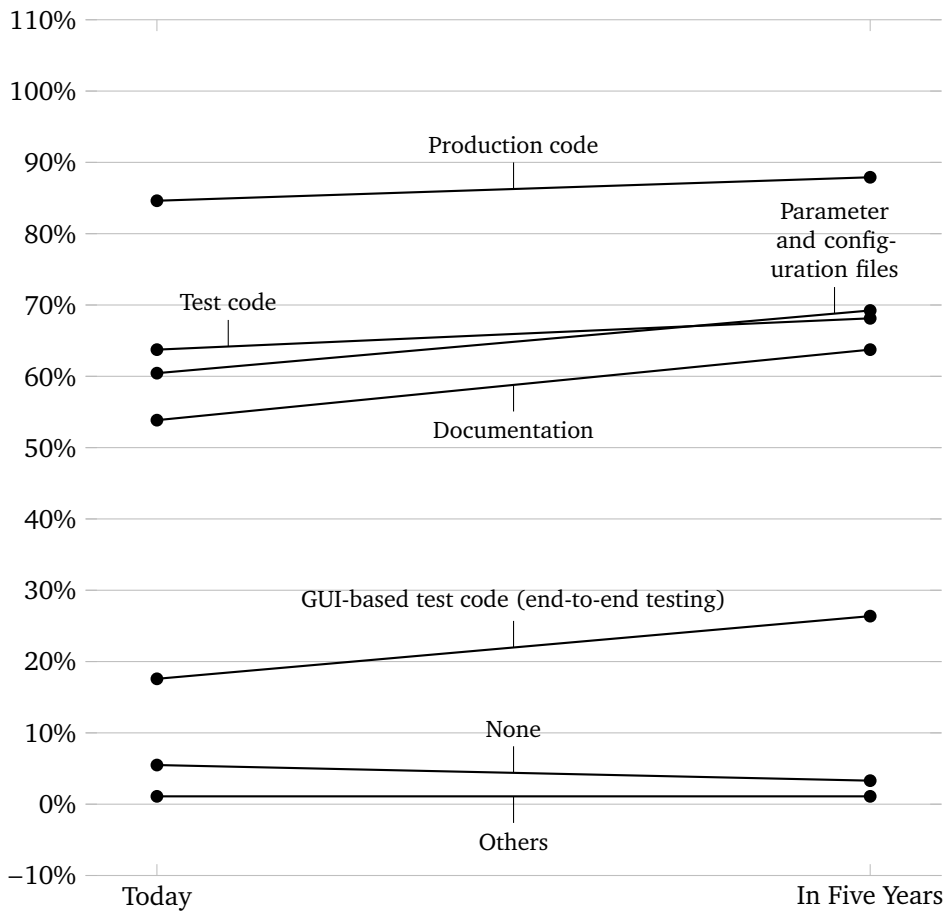


Figure 7.2: Practitioners anticipate reviewing a broader range of artifacts in the future, with notable growth in attention to GUI-based test code and a decline in those reviewing no artifacts at all.

Alternatively, LLMs could serve as the authors, *automatically generating code* that is then reviewed by humans, keeping developers "in the loop": "*Code will be generated by generative-AI based tools, but reviewed by humans. A review by humans will be mandatory to avoid any law suits on the code development organization.*" Again, the depth of involvement in code authoring lies on a spectrum, from assisting developers to autonomously proposing and integrating code changes. The amount and pace of generated code by AI will force us to review the code in a different way. In this scenario, concerns arise about the long-term impact on developer expertise. As one participant warned: "*I am a bit concerned about the future generation of developers as they will have less experience actually coding.*"

Two respondents also considered that automated code review intersects with automated code generation and "*AI [is] everywhere*". We refer to the intersection where both code generation and review are fully handled by LLMs and eliminating the need for human involvement as *unsupervised software engineering*. Yet this shift comes with caveats, as one participant explicitly warned:

> *"We will have to decide if we want to write code with AI or review code with AI [...] I don't think both at the same time will work, because there can be instances, where the programmer generates bad code with the AI, and a mistake is kept in, and because of time shortage the reviewer also uses AI to review the same code, and the AI doesn't catch the mistake because it generated it."*

Expressing concern about the future towards unsupervised software engineering, one developer stated: "*I expect that there will be more generated code of a worsen quality. I expect that there will be a lot of AI code review systems, which will be pretty useless.*"

We also encountered fundamental opposition to the use of AI in code review—whether as author or reviewer—as one participant put it: "*As long as AI is involved, it is likely to be a more tedious process [code review] than before.*" We refer to this stance as human-led software engineering, a position in direct contrast to a future of fully unsupervised software engineering without human oversight.

Based on practitioner input, we summarize the anticipated roles of AI in code review as a spectrum of involvement, gradually ranging from fully human-led to fully LLM-led, in both the roles of code author and reviewer. As illustrated in Figure 7.3, four possible futures of software engineering emerge from this gradual interplay between humans and LLMs, each reflecting a distinct balance in the roles of author and reviewer.

7.4 Doesn't LGTM

Our survey highlights two key expectations from practitioners about the future of code review. First, code review will remain a central practice in collaborative software engineering. Developers anticipate that it will continue to demand the same or even

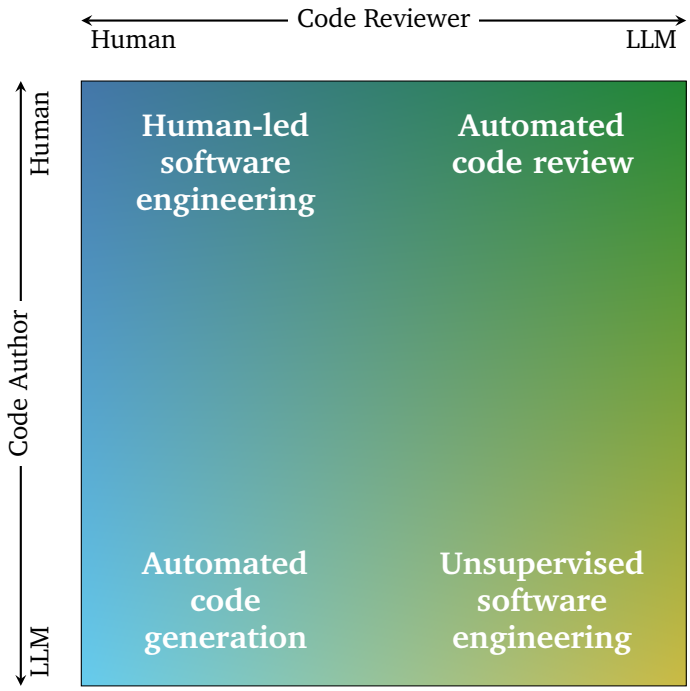


Figure 7.3: Practitioners expect the roles of code author and code reviewer to lie on continuous spectra, each ranging from fully human-led to fully LLM-led. Based on this human–AI interplay in code review, four possible futures of collaborative software engineering begin to take shape.

more time in the coming years, while expanding to include a broader range of artifacts. Second, with the growing integration of LLMs, the traditionally human-driven nature of code review with developers leading discussions, making judgments, and approving changes [8, 39] is expected to shift.

In this section, we invite the reader to envision the future that practitioners anticipate, one where AI takes a central role in code review. Based on this premise, we discuss three forms of erosion that are likely to affect collaborative software engineering broadly, but surface first in code review.

7.4.1 Erosion of Understanding

At its core, code review, and all the many review guidelines that support it [13], ultimately boils down to a single key question: *Would I be able to maintain this code change in the future if the original author were no longer available?* If the answer is no, it signals that the code may be too complex, poorly documented, badly structured, or unclear, and likely needs revision. By introducing a second pair of eyes, code review not only enhances maintainability and consistency but also acts as a mechanism for capturing and preserving human understanding of the software system and the broader implications of ongoing changes.

Now, imagine a future where an LLM handles code reviews and optimizes code to its own capacity to "understand" it. Ironically, this could lead to code becoming less maintainable over time—from a human perspective. What is readable and logical to an LLM may be opaque and not intuitive to a human developer. As a result, human understanding of the changes in the evolving codebase could gradually erode, leaving only other LLMs capable of maintaining it. And this might happen subtle: Even for AI-assisted code reviews where we keep the human in the process, reviewers focus more on the annotated code section and potentially overlooking issues elsewhere which could limit a comprehensive understanding [125].

Should we, then, make it a priority for humans to review code that is authored (partially or entirely) by LLMs to ensure long-term human maintainability of the codebase, not just short-term AI comprehension? This approach also has its cons. When developers routinely defer to LLM-generated solutions—especially without critically engaging with or questioning them—we risk fostering a *passive development culture*. Combined with an increased volume of LLM-generated code and thus review workload, developers may be forced to treat code as a black box—accepting solutions that seem to work, but without fully grasping their structure, intent, or long-term consequences.

Code is more than just functional software. It is a shared language among developers. If only machines can understand it, we risk losing not only our connection to the people we build systems for, but also our ability to meaningfully control the systems we create.

7.4.2 Erosion of Accountability

Code review is not just about finding bugs—it is about reinforcing accountability, ownership, and responsibility in software development [7]. By requiring developers to justify their changes and reviewers to critically evaluate them, code review ensures that every line of code has a clear owner—someone responsible for its quality, security, and long-term maintainability [134]. In contrast, AI systems can neither be held responsible nor accountable (let alone liable) in the same way, creating a gap in ownership of the codebase.

This gap places software companies in a legal limbo in the context of regulatory compliance. Emerging regulations such as the EU's *Cyber Resilience Act*, which mandates effective and regular reviews of software systems with a focus on security, or the *Transfer Pricing Guidelines* by the OECD, which call for a more precise definition of code ownership, both place accountability at the forefront of regulatory priorities and expectations [37].

As AI systems take on a greater role in collaborative software engineering, ensuring clear human accountability will be critical [107], not only for maintaining code quality, but also for meeting the growing legal and regulatory demands placed on software organizations.

7.4.3 Erosion of Trust

Trust is fundamental for code review—just as it is in any form of human collaboration. With the increasing involvement and reliance on LLMs, the foundation for trust is at risk. It is becoming increasingly unclear who truly authored a piece of code or a code review. Was it written by a human? By an LLM? Or, perhaps more worryingly, a human blindly parroting LLM-generated content without critical thought or understanding? This growing uncertainty signals a deeper issue—a gradual *erosion of trust* in the authenticity and accountability of the review process.

The erosion of trust in code review is not a distant threat—it is a problem we already face. In his blog post titled "The I in LLM stands for intelligence," Daniel Stenberg, the maintainer of the curl project, describes how some individuals use LLMs to analyze curl's code and generate security vulnerability reports [112]. These AI-generated reports often appear legitimate but are shallow and inaccurate, forcing maintainers to spend valuable time assessing and eventually dismissing them. This problem highlights the growing challenges to distinguish meaningful contributions from synthetic.

We fear a similar risk in code review. When reviewers cannot distinguish between genuine human insight and automated suggestions, feedback becomes superficial and potentially harmful. Code review loses its purpose and deviates from its critical function, becoming an empty shell—a routine for the sake of routine, rather than being a meaningful safeguard for code quality and collaboration.

7.5 Conclusion

Code review has long been at the heart of collaborative software engineering where experience, reasoning, and thoughtful feedback drive the evolution of software systems. And our results indicate that code review will not disappear in the foreseeable future. In fact, developers expect to spend the same or even more time reviewing code, and reviewing more artifacts in the process.

While code review will remain essential, it will evolve. The results of our survey clearly show that LLMs are expected to become ubiquitous and to take on more tasks. The question remains to what extent code review shall be delegated to AI or kept in human hands? Without unnecessarily demonizing the LLMs, we raise concerns for the consequences of routine over-reliance on LLMs in code review—risks of eroding human understanding, accountability, and trust, ultimately reducing code review to an empty shell.

As software engineering is in constant evolution, raising questions about how to improve efficiency through automation, we advocate for a nuanced approach to code review. Human understanding (even if not very efficient), accountability (even if not on today's agenda), and trust (even if not easily measured) ensure that we keep control of the long-term evolution of our software systems and remain resilient in the face of future changes, whether technical or transformational.

We advocate for further research to explore the implications of integrating LLMs and other AI technologies into code review, as we believe code review is where the effects of LLM adoption in collaborative software engineering are likely to surface first. An over-reliance on AI gradually shifts the focus of understanding from human teams to the LLM—an entity that, despite its capabilities, cannot be held accountable, reason contextually, or justify its decisions beyond probabilistic associations. To preserve the integrity of software engineering as a collaborative, transparent, trustworthy, and responsible engineering practice, we must ensure that code review remains grounded in human understanding and oversight.

Acknowledgement

We thank all participants of the survey. This work was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

References

[1] Adam Alami, Marisa Leavitt Cohn, and Andrzej Wasowski. "*Why Does Code Review Work for Open Source Software Communities?*" In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, May 2019, pp. 1073–1083. DOI: 10.1109/ICSE.2019.00111.

[2] Isabel Anger and Christian Kittl. "*Measuring influence on Twitter*". In: *Proceedings of the 11th International Conference on Knowledge Management and Knowledge Technologies*. ACM, September 2011, pp. 1–4. DOI: 10.1145/2024288.2024326.

[3] Alessia Antelmi, Gennaro Cordasco, Carmine Spagnuolo, and Vittorio Scarano. "*A design-methodology for epidemic dynamics via time-varying hypergraphs*". In: *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS* 2020-May (2020), pp. 61–69. DOI: 10.5555/3398761.3398774.

[4] Foundjem Armstrong, Foutse Khomh, and Bram Adams. "*Broadcast vs. Unicast Review Technology: Does It Matter?*" In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, March 2017, pp. 219–229. DOI: 10.1109/ICST.2017.27.

[5] Ikram El Asri, Noureddine Kerzazi, Gias Uddin, Foutse Khomh, and M.A. Janati Idrissi. "*An empirical study of sentiments in code reviews*". In: *Information and Software Technology* 114 (October 2019), pp. 37–54. DOI: 10.1016/j.infsof.2019.06.005.

[6] Claudia Ayala, Burak Turhan, Xavier Franch, and Natalia Juristo. "*Use and Misuse of the Term "Experiment" in Mining Software Repositories Research*". In: *IEEE Transactions on Software Engineering* 48 (11 November 2022), pp. 4229–4248. DOI: 10.1109/TSE.2021.3113558.

[7] Alberto Bacchelli and Christian Bird. "*Expectations, outcomes, and challenges of modern code review*". In: *Proceedings - International Conference on Software Engineering* (2013), pp. 712–721. DOI: 10.1109/ICSE.2013.6606617.

[8] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. "*Modern Code Reviews - A Survey of Literature and Practice*". In: *ACM Transactions on Software Engineering and Methodology* (February 2023). DOI: 10.1145/3585004.

[9] Sebastian Baltes and Paul Ralph. "*Sampling in software engineering research: a critical review and guidelines*". In: *Empirical Software Engineering* 27 (4 July 2022), p. 94. DOI: 10.1007/s10664-021-10072-8.

[10] Jerry Banks, J.S. Carson, Barry L Nelson, and David M Nicol. *Discrete event system simulation Solutions Manual*. 5th ed. Pearson Education, 2010, p. 639.

[11] Ann Barcomb, Klaas-Jan Stol, Brian Fitzgerald, and Dirk Riehle. "*Managing Episodic Volunteers in Free/Libre/Open Source Software Communities*". In: *IEEE Transactions on Software Engineering* 5589 (2020), pp. 1–1.

[12] Ann Barcomb, Klaas-Jan Stol, Dirk Riehle, and Brian Fitzgerald. "*Why Do Episodic Volunteers Stay in FLOSS Communities?*" In: *2019 IEEE/ACM 41st*

International Conference on Software Engineering (ICSE). IEEE, May 2019, pp. 948–959. DOI: 10.1109/ICSE.2019.00100.

[13] Andreas Bauer, Riccardo Coppola, Emil Alégroth, and Tony Gorschek. "*Code review guidelines for GUI-based testing artifacts*". In: *Information and Software Technology* 163 (November 2023), p. 107299. DOI: 10.1016/j.infsof.2023.107299.

[14] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. "*Factors influencing code review processes in industry*". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. 2016. DOI: 10.1145/2950290.2950323.

[15] Gabriele Bavota and Barbara Russo. "*Four eyes are better than two: On the impact of code reviews on software quality*". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, September 2015, pp. 81–90. DOI: 10.1109/ICSM.2015.7332454.

[16] Amiangshu Bosu and Jeffrey C. Carver. "*Impact of developer reputation on code review outcomes in OSS projects*". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, September 2014, pp. 1–10. DOI: 10.1145/2652524.2652544.

[17] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. "*Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft*". In: *IEEE Transactions on Software Engineering* 43 (1 2017), pp. 56–75. DOI: 10.1109/TSE.2016.2576451.

[18] Amiangshu Bosu, Michaela Greiler, and Christian Bird. "*Characteristics of useful code reviews: An empirical study at Microsoft*". In: *IEEE International Working Conference on Mining Software Repositories*. Vol. 2015-Augus. 2015, pp. 146–156. DOI: 10.1109/MSR.2015.21.

[19] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, František Plášil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. "*What characterizes a (software) component?*" In: *Software - Concepts & Tools* 19 (1 March 1998), pp. 49–56. DOI: 10.1007/s003780050007.

[20] Maximilian Capraro, Michael Dorner, and Dirk Riehle. "*The patch-flow method for measuring inner source collaboration*". In: *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. ACM Press, 2018, pp. 515–525. DOI: 10.1145/3196398.3196417.

[21] Maximilian Capraro and Dirk Riehle. "*Inner Source Definition, Benefits, and Challenges*". In: *ACM Computing Surveys* 49 (4 December 2016), pp. 1–36. DOI: 10.1145/2856821.

[22] Marco Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. "*Heartbleed 101*". In: *IEEE Security & Privacy* 12 (4 July 2014), pp. 63–67. DOI: 10.1109/MSP.2014.66.

[23] Jeffrey C Carver. *Towards reporting guidelines for experimental replications: A proposal*. 2010.

[24] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. "*Time-varying graphs and dynamic networks*". In: *International Journal of Parallel, Emergent and Distributed Systems* 27.5 (October 2012), pp. 387–408. DOI: 10.1080/17445760.2012.668546.

[25] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. "*WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review*". In: *Applied Soft Computing* 100 (March 2021), p. 106908. DOI: 10.1016/j.asoc.2020.106908.

[26] Victoria Clarke and Virginia Braun. "*Thematic analysis*". In: *The Journal of Positive Psychology* 12 (3 May 2017), pp. 297–298. DOI: 10.1080/17439760.2016.1262613.

[27] Association for Computing Machinery (ACM). 2020. URL: https://www.acm.org/publications/badging-terms (visited on 04/04/2024).

[28] Margarita Cruz, Beatriz Bernárdez, Amador Durán, Jose A Galindo, and Antonio Ruiz-Cortés. "*Replication of studies in empirical software engineering: A systematic mapping study, from 2013 to 2018*". In: *IEEE Access* 8 (2019), pp. 26773–26791.

[29] Atacílio Cunha, Tayana Conte, and Bruno Gadelha. "*Code Review is just reviewing code? A qualitative study with practitioners in industry*". In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, September 2021, pp. 269–274. DOI: 10.1145/3474624.3477063.

[30] Nicole Davila, Jorge Melegati, and Igor Wiese. "*Tales From the Trenches: Expectations and Challenges From Practice for Code Review in the Generative AI Era*". In: *IEEE Software* 41 (6 November 2024), pp. 38–45. DOI: 10.1109/MS.2024.3428439.

[31] Michael Dorner. *michaeldorner/only-time-will-tell: v2.0*. Version v2.0. June 2022. DOI: 10.5281/zenodo.6719261.

[32] Michael Dorner. *Only Time Will Tell*. Version 2.0. Zenodo, May 2022. DOI: 10.5281/zenodo.6542540.

[33] Michael Dorner. *The Upper Bound of Information Diffusion in Code Review*. Version 1.1. Zenodo, June 2023. DOI: 10.5281/zenodo.8042256.

[34] Michael Dorner and Andreas Bauer. *michaeldorner/information-diffusion-boundaries-in- code-review: 1.0*. Version 1.0. December 2023. DOI: 10.5281/zenodo.10417852.

[35] Michael Dorner, Andreas Bauer, and Florian Angermeir. "*No Free Lunch: Research Software Testing in Teaching*". Manuscript under review at *Journal of Open Research Software*. 2024.

[36] Michael Dorner, Andreas Bauer, Darja Šmite, Lukas Thode, Daniel Mendez, Ricardo Britto, Stephan Lukasczyk, Ehsan Zabardast, and Michael Kormann. "*Quo Vadis, Code Review?*" Manuscript under review at *IEEE Software*. 2025.

[37] Michael Dorner, Maximilian Capraro, Oliver Treidler, Tom-Eric Kunz, Darja Šmite, Ehsan Zabardast, Daniel Mendez, and Krzysztof Wnuk. "*Taxing Collaborative Software Engineering*". In: *IEEE Software* (2024), pp. 1–8. DOI: 10.1109/MS.2023.3346646.

[38] Michael Dorner and Daniel Mendez. "*The Capability of Code Review as a Communication Network*". Manuscript currently under review at *Journal of Empirical Software Engineering*. 2025.

[39] Michael Dorner, Daniel Mendez, Krzysztof Wnuk, Ehsan Zabardast, and Jacek Czerwonka. "*The Upper Bound of Information Diffusion in Code Review*". In: *Empirical Software Engineering* (June 2023).

[40] Michael Dorner, Daniel Mendez, Ehsan Zabardast, Nicole Valdez, and Marcin Floryan. "*Measuring Information Diffusion in Code Review at Spotify*". Registered report accepted at *International Symposium on Empirical Software Engineering and Measurement (ESEM)*; manuscript currently under review at *Journal of Empirical Software Engineering*. 2025.

[41] Michael Dorner, Darja Šmite, Daniel Mendez, Krzysztof Wnuk, and Jacek Czerwonka. "*Only Time Will Tell: Modelling Information Diffusion in Code Review with Time-Varying Hypergraphs*". In: *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, September 2022, pp. 195–204. DOI: 10.1145/3544902.3546254.

[42] Michael Dorner, Oliver Treidler, Tom-Eric Kunz, Ehsan Zabardast, Daniel Mendez, Darja Šmite, Maximilian Capraro, and Krzysztof Wnuk. "*Describing Globally Distributed Software Architectures for Tax Compliance*". 2023.

[43] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. "*Communicative Intention in Code Review Questions*". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, September 2018, pp. 519–523. DOI: 10.1109/ICSME.2018.00061.

[44] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. "*Confusion Detection in Code Reviews*". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, September 2017, pp. 549–553. DOI: 10.1109/ICSME.2017.40.

[45] M. E. Fagan. "*Design and code inspections to reduce errors in program development*". In: *IBM Systems Journal* 15 (3 1976), pp. 182–211. DOI: 10.1147/sj.153.0182.

[46] Michael Felderer, Guilherme Horta, and Travassos Editors. *Contemporary Empirical Methods in Software Engineering*. Ed. by Michael Felderer and Guilherme Horta Travassos. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-32489-6.

[47] Brian Fitzgerald and Klaas-Jan Stol. "*Continuous software engineering: A roadmap and agenda*". In: *Journal of Systems and Software* 123 (January 2017), pp. 176–189. DOI: 10.1016/j.jss.2015.06.063.

[48] Breno Bernard Nicolau de França and Nauman Bin Ali. "*The Role of Simulation-Based Studies in Software Engineering Research*". In: *Contemporary Empirical Methods in Software Engineering*. Cham: Springer International Publishing, 2020, pp. 263–287. DOI: 10.1007/978-3-030-32489-6_10.

[49] Breno Bernard Nicolau de França and Guilherme Horta Travassos. "*Experimentation with dynamic simulation models in software engineering: planning and reporting guidelines*". In: *Empirical Software Engineering* 21.3 (June 2016), pp. 1302–1345. DOI: 10.1007/s10664-015-9386-4.

[50] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. "*A survey of graph edit distance*". In: *Pattern Analysis and Applications* 13 (1 February 2010), pp. 113–129. DOI: 10.1007/s10044-008-0141-y.

[51] Michaela Goetz, Jure Leskovec, Mary McGlohon, and Christos Faloutsos. "*Modeling Blog Dynamics*". In: *Proceedings of the International AAAI Conference on Web and Social Media* 3 (1 March 2009), pp. 26–33. DOI: 10.1609/icwsm.v3i1.13941.

[52] Omar S Gómez, Natalia Juristo, and Sira Vegas. "*Replications types in experimental disciplines*". In: *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*. 2010, pp. 1–10.

[53] Omar S. Gómez, Natalia Juristo, and Sira Vegas. "*Understanding replication of experiments in software engineering: A classification*". In: *Information and Software Technology* 56 (8 August 2014), pp. 1033–1048. DOI: 10.1016/j.infsof.2014.04.004.

[54] Jesus M. Gonzalez-Barahona and Gregorio Robles. "*Revisiting the reproducibility of empirical software engineering studies based on data retrieved from development repositories*". In: *Information and Software Technology* 164 (December 2023), p. 107318. DOI: 10.1016/j.infsof.2023.107318.

[55] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. "*Analysing Time-Stamped Co-Editing Networks in Software Development Teams using git2net*". In: *Empirical Software Engineering* 26 (4 July 2021), p. 75. DOI: 10.1007/s10664-020-09928-2.

[56] A A Hagberg, D A Schult, and P J Swart. "*Exploring network structure, dynamics, and function using NetworkX*". In: *7th Python in Science Conference (SciPy 2008)* SciPy (2008), pp. 11–15.

[57] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, A. E. Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. "*Who does what during a code review? Datasets of OSS peer review repositories*". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, May 2013, pp. 49–52. DOI: 10.1109/MSR.2013.6624003.

[58] Lo Heander, Emma Söderberg, and Christofer Rydenfält. "*Support, Not Automation: Towards AI-supported Code Review for Code Quality and Beyond*". In: *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23â•fi28, 2025, Trondheim, Norway*. Vol. 1. ACM, 2025. DOI: 10.1145/3696630.3728505.

[59] Bernd Helmig, Katharina Spraul, and Karin Tremp. "*Replication Studies in Nonprofit Research*". In: *Nonprofit and Voluntary Sector Quarterly* 41 (3 June 2012), pp. 360–385. DOI: 10.1177/0899764011404081.

[60] Steffen Herbold, Aynur Amirfallah, Fabian Trautsch, and Jens Grabowski. "*A systematic mapping study of developer social network research*". In: *Journal of Systems and Software* 171 (January 2021), p. 110802. DOI: 10.1016/j.jss.2020.110802.

[61] J.D. Herbsleb and D. Moitra. "*Global software development*". In: *IEEE Software* 18 (2 2001), pp. 16–20. DOI: 10.1109/52.914732.

[62] Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. "*The review linkage graph for code review analytics: a recovery approach and empirical study*". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, August 2019, pp. 578–589. DOI: 10.1145/3338906.3338949.

[63] Guilherme Horta and Travassos Márcio De Oliveira Barros. "*Contributions of In Virtuo and In Silico Experiments for the Future of Empirical Studies in Software Engineering*". In: *2nd Workshop on empirical software engineering the future of empirical studies in software engineering*. 2003, pp. 117–130.

[64] Theresia Devi Indriasari, Andrew Luxton-Reilly, and Paul Denny. "*A Review of Peer Code Review in Higher Education*". In: *ACM Transactions on Computing Education* 20 (3 September 2020), pp. 1–25. DOI: 10.1145/3403935.

[65] ISO/IEC. *International Vocabulary of Metrology–Basic and General Concepts and Associated Terms*. 2007.

[66] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. "*Reporting Experiments in Software Engineering*". In: *Guide to Advanced Empirical Software Engineering*. Ed. by Forrest Shull, Janice Singer, and Dag I. K. Sjøberg. Springer London, 2008, pp. 201–228. DOI: 10.1007/978-1-84800-044-5_8.

[67] Mitchell Joblin, Barbara Eckl, 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* 32 (4 July 2023), pp. 1–29. DOI: 10.1145/3569949.

[68] David Kavaler, Premkumar Devanbu, and Vladimir Filkov. "*Whom are you going to call? Determinants of @-mentions in Github discussions*". In: *Empirical Software Engineering* 24 (6 December 2019), pp. 3904–3932. DOI: 10.1007/s10664-019-09728-3.

[69] Hemank Lamba, Asher Trockman, Daniel Armanios, Christian Kästner, Heather Miller, and Bogdan Vasilescu. "*Heard it through the Gitvine: an empirical study of tool diffusion across the npm ecosystem*". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, November 2020, pp. 505–517. DOI: 10.1145/3368089.3409705.

[70] Amanda Lee and Jeffrey C. Carver. "*Are One-Time Contributors Different? A Comparison to Core and Periphery Developers in FLOSS Repositories*". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, November 2017, pp. 1–10. DOI: 10.1109/ESEM.2017.7.

[71] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. "*Meme-tracking and the dynamics of the news cycle*". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, June 2009, pp. 497–506. DOI: 10.1145/1557019.1557077.

[72] Michael Levandowsky and David Winter. "*Distance between Sets*". In: *Nature* 234 (5323 November 1971), pp. 34–35. DOI: 10.1038/234034a0.

[73] Lisha Li, Zhilei Ren, Xiaochen Li, Weiqin Zou, and He Jiang. "*How Are Issue Units Linked? Empirical Study on the Linking Behavior in GitHub*". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 2018-December. IEEE, December 2018, pp. 386–395. DOI: 10.1109/APSEC.2018.00053.

[74] David Liben-Nowell and Jon Kleinberg. "*Tracing information flow on a global scale using Internet chain-letter data*". In: *Proceedings of the National Academy of Sciences* 105 (12 March 2008), pp. 4633–4638. DOI: 10.1073/pnas.0708471105.

[75] Daniel Mendez, Paris Avgeriou, Marcos Kalinowski, and Nauman Bin Ali. "*Teaching Empirical Software Engineering: An Editorial Introduction*". In: *Handbook on Teaching Empirical Software Engineering*. Springer, 2024, pp. 3–12.

[76] Daniel Mendez, Daniel Graziotin, Stefan Wagner, and Heidi Seibold. "*Open Science in Software Engineering*". In: *Contemporary Empirical Methods in Software Engineering*. Springer International Publishing, 2020, pp. 477–501. DOI: 10.1007/978-3-030-32489-6_17.

[77] Daniel Méndez and Jan-Hendrik Passoth. "*Empirical software engineering: From discipline to interdiscipline*". In: *Journal of Systems and Software* 148 (February 2019), pp. 170–179. DOI: 10.1016/j.jss.2018.11.019.

[78] Audris Mockus and James D. Herbsleb. "*Expertise browser: a quantitative approach to identifying expertise*". In: *Proceedings of the 24th international conference on Software engineering - ICSE '02*. ACM Press, 2002, p. 503.

[79] Audris Mockus, Peter C Rigby, Rui Abreu, Anatoly Akkerman, Yogesh Bhootada, Payal Bhuptani, Gurnit Ghardhora, Lan Hoang Dao, Chris Hawley, Renzhi He, Sagar Krishnamoorthy, Sergei Krauze, Jianmin Li, Anton Lunov, Dragos Martac, Francois Morin, Neil Mitchell, Venus Montes, Maher Saba, Matt

Steiner, Andrea Valori, Shanchao Wang, and Nachiappan Nagappan. "*Code Improvement Practices at Meta*". In: (April 2025).

[80] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. "*Gerrit software code review data from Android*". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, May 2013, pp. 45–48. DOI: 10.1109/MSR.2013.6624002.

[81] Mark Müller and Dietmar Pfahl. "*Simulation Methods*". In: *Guide to Advanced Empirical Software Engineering*. London: Springer London, 2008, pp. 117–152. DOI: 10.1007/978-1-84800-044-5_5.

[82] Sumaira Nazir, Nargis Fatima, and Suriayati Chuprat. "*Modern Code Review Benefits-Primary Findings of A Systematic Literature Review*". In: *Proceedings of the 3rd International Conference on Software Engineering and Information Management*. ACM, January 2020, pp. 210–215. DOI: 10.1145/3378936.3378954.

[83] Roozbeh Nia, Christian Bird, Premkumar Devanbu, and Vladimir Filkov. "*Validity of network analyses in Open Source Projects*". In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, May 2010, pp. 201–209. DOI: 10.1109/MSR.2010.5463342.

[84] Vincenzo Nicosia, John Tang, Mirco Musolesi, Giovanni Russo, Cecilia Mascolo, and Vito Latora. "*Components in time-varying graphs*". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 22.2 (June 2012), p. 023101. DOI: 10.1063/1.3697996.

[85] M.E. Nordberg. "*Managing code ownership*". In: *IEEE Software* 20 (2 March 2003), pp. 26–33. DOI: 10.1109/MS.2003.1184163.

[86] OECD. *Addressing the Tax Challenges of the Digital Economy, Action 1 - 2015 Final Report*. 2015, p. 288. DOI: 10.1787/9789264241046-en.

[87] OECD. *OECD Transfer Pricing Guidelines for Multinational Enterprises and Tax Administrations 2022*. 2022, p. 659. DOI: 10.1787/0e655865-en.

[88] Marcel Olbert and Christoph Spengel. "*International Taxation in the Digital Economy: Challenge Accepted?*" In: *World tax journal* 9.1 (2017), pp. 3–46.

[89] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. "*Search-Based Peer Reviewers Recommendation in Modern Code Review*". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, October 2016, pp. 367–377. DOI: 10.1109/ICSME.2016.65.

[90] Xavier Ouvrard. "*Hypergraphs: an introduction and review*". February 2020. DOI: 10.48550/arXiv.2002.05014.

[91] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. "*An Empirical Study of the Non-determinism of ChatGPT in Code Generation*". In: *ACM Transactions on Software Engineering and Methodology* (September 2024). DOI: 10.1145/3697010.

[92] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. "*Information Needs in Contemporary Code Review*". In: *Proceedings of*

the ACM on Human-Computer Interaction 2 (CSCW November 2018), pp. 1–27. DOI: 10.1145/3274404.

[93] Vesa Peltokorpi. "*Transactive memory systems*". In: *Review of general Psychology* 12.4 (2008), pp. 378–394.

[94] Karl Popper. *The Logic of Scientific Discovery*. Julius Springer, Hutchinson & Co, 1959.

[95] Rachel Potvin and Josh Levenberg. "*Why Google stores billions of lines of code in a single repository*". In: *Communications of the ACM* 59 (June 2016), pp. 78–87.

[96] Evan Priestley. *Phacility is Winding Down Operations*. Last access on 16.05.2023. 2021.

[97] Dirk Riehle. "*The Future of the Open Source Definition*". In: *Computer* 56 (12 December 2023), pp. 95–99. DOI: 10.1109/MC.2023.3311648.

[98] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. "*Contemporary Peer Review in Action: Lessons from Open Source Development*". In: *IEEE Software* 29 (6 November 2012), pp. 56–61. DOI: 10.1109/MS.2012.24.

[99] Peter C. Rigby and Christian Bird. "*Convergent contemporary software peer review practices*". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. 2013, p. 202. DOI: 10.1145/2491411.2491444.

[100] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. "*Open Source Software Peer Review Practices: A Case Study of the Apache Server*". In: *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, 2008, p. 541. DOI: 10.1145/1368088.1368162.

[101] Peter C. Rigby and Margaret-Anne Storey. "*Understanding broadcast based peer review on open source software projects*". In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, May 2011, pp. 541–550. DOI: 10.1145/1985793.1985867.

[102] Nicolli Rios, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola. "*A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners*". In: *Information and Software Technology* 102 (October 2018), pp. 117–145. DOI: 10.1016/j.infsof.2018.05.010.

[103] Stewart Robinson. "*Conceptual Modeling for Simulation: Issues and Research Requirements*". In: *Proceedings of the 2006 Winter Simulation Conference*. 1994. IEEE, December 2006, pp. 792–800. DOI: 10.1109/WSC.2006.323160.

[104] Everett M. Rogers. *Diffusion of Innovations*. New York: Free Press, August 2003, p. 576.

[105] Leonore Röseler, Ingo Scholtes, and Christoph Gote. "*A Network Perspective on the Influence of Code Review Bots on the Structure of Developer Collaborations*". April 2023. DOI: 10.48550/arXiv.2304.14787.

[106] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. "*Modern Code Review: A Case Study at Google*". In: *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18*. 2018, pp. 181–190.

[107] Jaakko Sauvola, Sasu Tarkoma, Mika Klemettinen, Jukka Riekki, and David Doermann. "*Future of software development with generative AI*". In: *Automated Software Engineering* 31.1 (2024). DOI: 10.1007/s10515-024-00426-z.

[108] Forrest J. Shull, Jeffrey C. Carver, Sira Vegas, and Natalia Juristo. "*The role of replications in Empirical Software Engineering*". In: *Empirical Software Engineering* 13 (2 April 2008), pp. 211–218. DOI: 10.1007/s10664-008-9060-1.

[109] Dag I. K. Sjøberg, Tore Dybå, Bente C. D. Anda, and Jo E. Hannay. "*Building Theories in Software Engineering*". In: *Guide to Advanced Empirical Software Engineering*. Springer London, 2008, pp. 312–336. DOI: 10.1007/978-1-84800-044-5_12.

[110] Darja Šmite, Nils Brede Moe, Marcin Floryan, Javier Gonzalez-Huerta, Michael Dorner, and Aivars Sablis. "*Decentralized decision-making and scaled autonomy at Spotify*". In: *Journal of Systems and Software* 200 (June 2023), p. 111649. DOI: 10.1016/j.jss.2023.111649.

[111] Darja Smite, Nils Brede Moe, Jarle Hildrum, Javier Gonzalez Huerta, and Daniel Mendez. "*Work-from-home is here to stay: Call for flexibility in post-pandemic work policies*". In: *Journal of Systems and Software* (January 2022), p. 111552. DOI: 10.1016/j.jss.2022.111552.

[112] Daniel Stenberg. *The I in LLM Stands For Intelligence*. 2024. URL: https://daniel.haxx.se/blog/2024/01/02/the-i-in-llm-stands-for-intelligence/ (visited on 05/29/2025).

[113] Klaas-Jan Stol. "*Teaching Theorizing in Software Engineering Research*". In: *Handbook on Teaching Empirical Software Engineering*. Springer Nature Switzerland, 2024, pp. 31–69. DOI: 10.1007/978-3-031-71769-7_3.

[114] Klaas-Jan Stol and Brian Fitzgerald. "*The ABC of Software Engineering Research*". In: *ACM Transactions on Software Engineering and Methodology* 27 (September 2018), pp. 1–51.

[115] Klaas-Jan Stol and Brian Fitzgerald. "*Theory-oriented software engineering*". In: *Science of Computer Programming* 101 (April 2015), pp. 79–98. DOI: 10.1016/j.scico.2014.11.010.

[116] Anton Strand, Markus Gunnarson, Ricardo Britto, and Muhmmad Usman. "*Using a context-aware approach to recommend code reviewers*". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM, June 2020, pp. 1–10. DOI: 10.1145/3377813.3381365.

[117] Charles Teddlie. "*Mixed Methods Sampling: A Typology With Examples*". In: *Journal of Mixed Methods Research* 1 (2009), pp. 77–100.

[118] Patanamon Thongtanunam and Ahmed E. Hassan. "*Review Dynamics and Their Impact on Software Quality*". In: *IEEE Transactions on Software Engineering* 47 (12 December 2021), pp. 2698–2712. DOI: 10.1109/TSE.2020.2964660.

[119] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. "*Review participation in modern code review*". In: *Empirical Software Engineering* 22 (2 April 2017), pp. 768–817. DOI: 10.1007/s10664-016-9452-6.

[120] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. "*AutoTransform*". In: *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022, pp. 237–248. DOI: 10.1145/3510003.3510067.

[121] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. "*Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review*". In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, March 2015, pp. 141–150. DOI: 10.1109/SANER.2015.7081824.

[122] F. Thung, T. F. Bissyande, D. Lo, and Lingxiao Jiang. "*Network Structure of Social Coding in GitHub*". In: *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, March 2013, pp. 323–326. DOI: 10.1109/CSMR.2013.41.

[123] Oliver Treidler. *Transfer Pricing in One Lesson*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-25085-0.

[124] Oliver Treidler, Tom-Eric Kunz, Michael Dorner, and Maximilian Capraro. "*The Microsoft Case: Lessons for Post-BEPS Software Development Cost Contribution Arrangements*". In: *Tax Notes International* 114 (June 2024), pp. 1883–1894.

[125] Rosalia Tufano, Alberto Martin-Lopez, Ahmad Tayeb, Ozren Dabic, Sonia Haiduc, and Gabriele Bavota. " *Deep Learning-based Code Reviews: A Paradigm Shift or a Double-Edged Sword? "* In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 597–597. DOI: 10.1109/ICSE55347.2025.00060.

[126] Mairieli Wessel, Igor Wiese, Igor Steinmacher, and Marco Aurelio Gerosa. "*Don't Disturb Me: Challenges of Interacting with Software Bots on Open Source Software Projects*". In: *Proceedings of the ACM on Human-Computer Interaction* 5 (October 2021). DOI: 10.1145/3476042.

[127] Laurie Williams. "*Integrating pair programming into a software development process*". In: *Proceedings 14th Conference on Software Engineering Education and Training. 'In search of a software engineering profession' (Cat. No.PR01059)*. IEEE Comput. Soc, pp. 27–36. DOI: 10.1109/CSEE.2001.913816.

[128] Claes Wohlin. "*An Evidence Profile for Software Engineering Research and Practice*". In: *Perspectives on the Future of Software Engineering*. Springer Berlin Heidelberg, 2013, pp. 145–157. DOI: 10.1007/978-3-642-37395-4_10.

[129] Claes Wohlin and Austen Rainer. "*Challenges and recommendations to publishing and using credible evidence in software engineering*". In: *Information and Software Technology* 134 (June 2021), p. 106555. DOI: 10.1016/j.infsof.2021.106555.

[130] Yulin Xu and Minghui Zhou. "*A multi-level dataset of linux kernel patchwork*". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, May 2018, pp. 54–57. DOI: 10.1145/3196398.3196475.

[131] Xin Yang, Raula Gaikovina Kula, Camargo Cruz Ana Erika, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, and Hajimu Iida. "*Understanding OSS peer review roles in peer review social network (PeRSoN)*". In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC* 1 (2012), pp. 709–712. DOI: 10.1109/APSEC.2012.63.

[132] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. "*Mining the modern code review repositories*". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, May 2016, pp. 460–463. DOI: 10.1145/2901739.2903504.

[133] Ehsan Zabardast, Julian Frattini, Javier Gonzalez-Huerta, Daniel Mendez, Tony Gorschek, and Krzysztof Wnuk. "*Assets in Software Engineering: What are they after all?*" In: *Journal of Systems and Software* 193 (November 2022), p. 111485. DOI: 10.1016/j.jss.2022.111485.

[134] Ehsan Zabardast, Javier Gonzalez-Huerta, and Binish Tanveer. "*Ownership vs Contribution: Investigating the Alignment Between Ownership and Contribution*". In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, March 2022, pp. 30–34. DOI: 10.1109/ICSA-C54293.2022.00013.

[135] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. "*Automatically Recommending Peer Reviewers in Modern Code Review*". In: *IEEE Transactions on Software Engineering* 42 (6 June 2016), pp. 530–543. DOI: 10.1109/TSE.2015.2500238.

[136] Yang Zhang, Gang Yin, Yue Yu, and Huaimin Wang. "*A Exploratory Study of @-Mention in GitHub's Pull-Requests*". In: *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 1. IEEE, December 2014, pp. 343–350. DOI: 10.1109/APSEC.2014.58.

[137] Wisam Haitham Abbood Al-Zubaidi, Patanamon Thongtanunam, Hoa Khanh Dam, Chakkrit Tantithamthavorn, and Aditya Ghose. "*Workload-aware reviewer recommendation using a multi-objective search-based approach*". In: *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, November 2020, pp. 21–30. DOI: 10.1145/3416508.3417115.

This page intentionally left silly.