

IDENTIFYING KEY DEVELOPERS IN
SOFTWARE PROJECTS USING CODE
AUTHORSHIP METRICS

GUILHERME AMARAL AVELINO

IDENTIFYING KEY DEVELOPERS IN
SOFTWARE PROJECTS USING CODE
AUTHORSHIP METRICS

Tese apresentada ao Programa de Pós-Graduação em Computer Science do Instituto de Ciências Exatas da Federal University of Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Computer Science.

ORIENTADOR: MARCO TÚLIO DE OLIVEIRA VALENTE

Belo Horizonte

Junho de 2018

GUILHERME AMARAL AVELINO

IDENTIFYING KEY DEVELOPERS IN
SOFTWARE PROJECTS USING CODE
AUTHORSHIP METRICS

Thesis presented to the Graduate Program
in Computer Science of the Federal Univer-
sity of Minas Gerais in partial fulfillment of
the requirements for the degree of Doctor
in Computer Science.

ADVISOR: MARCO TÚLIO DE OLIVEIRA VALENTE

Belo Horizonte

June 2018

© 2018, Guilherme Amaral Avelino.
Todos os direitos reservados.

Avelino, Guilherme Amaral

A948i Identifying key developers in software projects using
code authorship metrics / Guilherme Amaral Avelino.
— Belo Horizonte, 2018
xxiii, 114 f. : il. ; 29cm

Tese (doutorado) — Federal University of Minas
Gerais

Orientador: Marco Túlio de Oliveira Valente

1. Computação - Teses. 2. Engenharia de software -
Desenvolvimento. 3. code authorship. 4. key developers.
5. truck factor. I. Título.

CDU 519.6*32.(043)



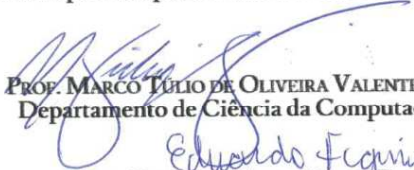
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

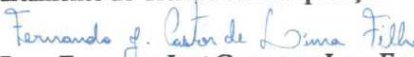
Identifying key developers in software projects using code authorship
metrics

GUILHERME AMARAL AVELINO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG


PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG


PROF. FERNANDO JOSÉ CASTOR DE LIMA FILHO
Centro de Informática - UFPE


PROF. MIRELLA MOURA MORO
Departamento de Ciência da Computação - UFMG


PROF. UIRA KULESZA
Departamento de Informática e Matemática Aplicada - UFRN

Belo Horizonte, 21 de junho de 2018.

To my wife, son and parents.

Acknowledgments

I would like to express my gratitude to everyone who helped me scientifically and emotionally along the long road that is a doctoral degree. I thank God for having all these people in my life and for granting me the capability to proceed successfully.

My advisor, Prof. Dr. Marco Tulio Valente, deserves a special acknowledgment. In the past four years, he provided me a mix of encouragement and challenge that guided me throughout my academic growth. I am very grateful for the time, wisdom and immense support that he dedicated to me on this journey. I could not have imagined having a better mentor for my Ph.D.

Besides my advisor, many others contributed to the research presented in this thesis. First, I would like to thank Prof. Dr. Alexander Serebrenik for giving me the opportunity to work under his supervision at TU/Eindhoven. The inspiring meetings and the new ideas resulting from this interaction were of great value to this work. Second, I must thank Leonardo Passos, Andre Hora, Fábio Petrillo and Eleni Constantinou for their collaboration on the papers and discussions that helped to mold the contributions of this thesis. Without your cooperation and effort, many parts of this thesis would not have been possible.

I also thank the members of the ASERG research group for their friendship and technical collaboration. More than colleagues I found friends there. In particular, I would like to express my gratitude to Cristiano Maffort and his family for welcoming me into their home, providing me help when I started this journey. Undoubtedly, they made the first days far from my family too much easier.

I thank the remaining members of my Ph.D. committee for reviewing this thesis and your insightful comments: Profa. Dra. Mirella Moura Moro, Prof. Dr. Eduardo Magno Lages Figueiredo, Prof. Dr. Fernando José Castor de Lima Filho, and Prof. Dr. Uira Kulesza.

I thank the Federal University of Piauí for permitting me to dedicate to my Ph.D. and also the colleagues from the Computer Department for sharing the burden of my absence.

Finally, I must express my very profound gratitude to my family. I would like to thank my parents, Paulo e Aldênia, whose love and guidance are with me in whatever I pursue. Additionally, my siblings, David e Marcela, and two angels that God put in my life, Maria Dacy and Noemy. Their love, incentive, and prayers mean a lot to me. Most importantly, I wish to thank my beloved wife and son. Lyvia and Matheus, I love you more than anything and I thank God for the gift of having you in my life. Thanks for comprehending and loving me even when I was far or hidden in the studies. No matter what I am doing, I am always thinking of you.

Resumo

Autoria de código é uma informação importante sobre grandes sistemas de software. Ela pode ser usada para investigar a divisão do trabalho, identificar colaboradores importantes e avaliar perfis dos desenvolvedores, entre outros. No entanto, seu uso prático no desenvolvimento de software ainda não é amplamente explorado. Para investigar este problema, nesta tese propomos e avaliamos, através de estudos quantitativos e qualitativos, aplicações práticas de autoria de código no desenvolvimento de software. Inicialmente, definimos um conjunto de conceitos centrados em autoria, que usamos para investigar as equipes de desenvolvimento de 115 projetos de código aberto, incluindo uma análise aprofundada do kernel do Linux. Depois disso, usamos métricas de autoria para abordar dois problemas bem conhecidos de Engenharia de Software: (1) avaliar a concentração de conhecimento em projetos de software e (2) identificar desenvolvedores qualificados para manter arquivos de código fonte específicos. Para resolver o primeiro problema, propomos um novo algoritmo para estimar *truck factors* (TF), uma métrica popular para revelar membros essenciais em um projeto. Usamos esse algoritmo para estimar o TF de 133 projetos e validamos os resultados entrevistando os desenvolvedores. Também aplicamos esse algoritmo para identificar eventos TF—isto é, situações em que todos os desenvolvedores TF abandonam o projeto—em um conjunto de 1.932 projetos populares do GitHub. Neste estudo, identificamos eventos TF em 315 projetos (16%) e observamos que 128 deles (41%) sobreviveram ao seu mais recente evento, ou seja, novos desenvolvedores assumiram o desenvolvimento do projeto. Em seguida, entrevistando os desenvolvedores, relatamos as práticas de programação que ajudaram esses projetos a superar tais eventos. Finalmente, para abordar o segundo problema, investigamos a eficácia de métricas de autoria de código para identificar mantenedores de código em 10 projetos (8 de código aberto e 2 comerciais). Os resultados revelam as limitações das técnicas existentes e fornecem orientações sobre como melhorá-las, controlando dados sobre tamanho do código e recência.

Palavras-chave: Autoria de código, desenvolvedores chave, truck factor.

Abstract

Code authorship is a key information about large software systems. It can be used to reason about the division of work in software projects, to identify key collaborators, and to assess developers' profiles, among others. However, its practical usage in software development is not widely explored. To tackle this problem, in this thesis, we propose and evaluate, through a set of quantitative and qualitative studies, practical applications of code authorship in software development. First, we define several authorship-centric concepts, which we use to investigate the development teams of 115 open source projects, including an in-depth analysis of the Linux kernel. After that, we use code authorship metrics to address two well-known software engineering problems: (1) assess knowledge concentration in software projects, and (2) identify skilled developers to maintain specific source code files. To address the first problem, we propose a novel algorithm to estimate truck factors (TF), a popular metric to reveal essential project members. We use this algorithm to estimate the TF of 133 projects and validate the results by surveying the systems' developers. We also apply this algorithm to identify TF events—i.e., situations where all TF developers abandon the project—in a set of 1,932 popular GitHub projects. In this study, we identified TF events in 315 projects (16%) and observed that 128 of them (41%) survived their most recent TF event, i.e., new developers assumed the project development. Then, by surveying the systems' developers, we report the programming practices that helped these projects to overcome such events. Finally, to address the second problem, we investigate the effectiveness of code authorship metrics to identify skilled source code maintainers in 10 projects (8 open source and 2 commercial). The results reveal the limitations of existing techniques and provide insights on how to improve them by controlling code size and recency data.

Keywords: Code authorship, key developers, truck factor.

List of Figures

2.1	Authors and developers over time	14
2.2	Distribution of the number of files per author in each release	16
2.3	Percentage of files authored by the top-10 authors over time	17
2.4	Gini coefficients	17
2.5	Number of files per author (release v4.7)	18
2.6	Specialists and generalists over time	19
2.7	Percentage of specialists and generalists	21
2.8	Fragment of the Linux co-authorship network	23
2.9	Co-authorship network properties over time	25
2.10	Authorship measures in an extended dataset of 114 open source systems .	28
2.11	Co-authorship measures in an extended dataset of 114 open source systems	29
3.1	Proposed approach for truck factor calculation	38
3.2	Target subjects	43
3.3	Respondents profile	46
3.4	Proportion of developers ranked as authors	47
3.5	Systems Truck Factor	48
3.6	Percentage of files per author in excilys/androidannotations (top-4 authors) . .	50
4.1	TF event on <i>composer/satis</i>	61
4.2	Surviving on a TF event on <i>composer/satis</i>	62
4.3	Number of projects by language.	63
4.4	Distribution of the number of developers, commits, files, and stars.	64
4.5	Percentage of aliases in each project	65
4.6	TF of the 1,932 projects in our dataset	65
4.7	Projects facing TF events	67
4.8	Contributions to PointCloudLibrary/pcl over time (screenshot from GitHub)	67
4.9	Age of the repositories with TF events	68

4.10	When do TF events happen (counting from the repositories creation) . . .	69
4.11	When do projects survive a TF event	69
4.12	Number of commits after the last observed TF events	70
4.13	Percentage of commits after the last observed TF events	70
4.14	Number of developers, commits and files, for surviving and non-surviving projects (at the date of the studied TF events)	71
5.1	Steps to construct the authorship oracle	86
5.2	Distribution of the survey answers	88
5.3	Computing HR_M , $HR_{\overline{M}}$, and HM for a hypothetical file f	90
5.4	Harmonic mean (HM)	91
5.5	Distribution of the percentage of commits by a developer d who is a <i>declared maintainer</i> of a file f when all techniques failed to identify this condition (<i>AllMiss</i>) and when all techniques correctly identified this condition (<i>AllHit</i>).	92
5.6	Distribution of <i>declared maintainers</i> and <i>declared non-maintainers</i> according to the following factors: <i>recency</i> (in days from the last developer commit) and <i>file size</i> (in LOC).	93
5.7	Harmonic mean (HM) of the hit ratio of <i>maintainers</i> and <i>non-maintainers</i> achieved when controlling for file size and recency	95

List of Tables

2.1	Linux subsystems (release v4.7)	11
2.2	Author Ratio Evolution: Summary Statistics. Avg: Average, Std Dev: Standard Deviation	14
2.3	Percentage of files with multiple authorship (release v4.7)	22
2.4	Co-authorship network properties (release v4.7)	23
2.5	Number of files authored by Solitary Authors (i.e., authors that do not have co-authors)	24
3.1	Target repositories	42
3.2	Systems with highest truck factors	48
3.3	Answers for Survey Question 1	49
3.4	Answers for Survey Question 2	51
3.5	Practices to attenuate the truck factor	52
4.1	Did you perceive the projects at risk?	74
4.2	Motivations to contribute	74
4.3	Characteristics that helped new TF developers	75
4.4	Barriers faced by new TF developers	76

Contents

Acknowledgments	xi
Resumo	xiii
Abstract	xv
List of Figures	xvii
List of Tables	xix
1 Introduction	1
1.1 Motivation and Problem	1
1.2 Proposed Thesis	2
1.3 Outline	5
2 Measuring and Analyzing Code Authorship	7
2.1 Introduction	7
2.2 Study Design	9
2.2.1 Author Identification	9
2.2.2 Linux Kernel Architectural Decomposition	11
2.2.3 Data Collection	11
2.2.4 Custom-made Infrastructure	13
2.3 Results	13
2.4 Discussion	26
2.5 Measuring Code Authorship in a Large Dataset	27
2.6 Threats to Validity	30
2.7 Related Work	31
2.8 Conclusion	33
3 Estimating Truck Factors	35

3.1	Introduction	35
3.2	Truck Factor: An Example from the Early Days of Python	37
3.3	Proposed Approach	38
3.3.1	Main Steps	39
3.4	Validation Methodology	40
3.4.1	Selection of Target Subjects	41
3.4.2	Setting up Inputs	42
3.4.3	Survey Design and Application	44
3.5	Truck Factor Estimates	46
3.5.1	Preceding Output	46
3.5.2	Results	47
3.6	TF Validation: Surveying Developers	49
3.7	Discussion	53
3.7.1	DOA Results	53
3.7.2	Challenges on Computing Truck Factors	53
3.8	Threats to Validity	54
3.9	Related Work	55
3.10	Conclusion	56
4	Investigating Truck Factor Events	57
4.1	Introduction	57
4.2	Truck Factor	60
4.2.1	Definitions	60
4.2.2	Identifying Truck Factor Events	60
4.2.3	Identifying Surviving Projects	61
4.3	Study Design	62
4.3.1	Dataset & Preprocessing	62
4.3.2	Aliases Handling	64
4.3.3	Estimating Truck Factors	65
4.4	Searching for TF Events and Surviving Projects	66
4.5	Survey with TF Developers	72
4.5.1	Survey Design	72
4.6	Discussion	77
4.7	Threats to validity	78
4.8	Related work	79
4.9	Conclusion	80

5	Identifying Software Maintainers	83
5.1	Introduction	83
5.2	Evaluated Techniques	84
5.3	Study Design	85
5.3.1	Target Systems	85
5.3.2	Oracle	86
5.3.3	Inferring Maintainers and Non-maintainers	88
5.3.4	Evaluation Metrics	88
5.4	Results	90
5.5	Discussion	91
5.5.1	When and Why the Techniques Fail	91
5.5.2	Controlling for File Size and Recency	94
5.6	Threats to Validity	95
5.7	Conclusion	96
6	Conclusion	97
6.1	Contributions	97
6.2	Discussion	99
6.2.1	Code Authorship Applicability	99
6.2.2	Truck Factor is a Real Risk in Open Source	99
6.3	Future Work	100
	Bibliography	103

Chapter 1

Introduction

1.1 Motivation and Problem

Software engineering is a collective effort, requiring the coordination of large development teams [Crowston and Howison, 2005; Herbsleb, 2007; Mistrík et al., 2010]. Furthermore, to tackle complexity and size, software systems are usually partitioned in components and sub-components, which ideally can be implemented in parallel [Parnas, 1972]. These key characteristics of software development projects—collaborative work and modularization—increase the value of code authorship information, which can be used to identify the best developers for a maintenance task, characterize developer’s profiles, and analyze development teams.

Essentially, the goal of measuring code authorship is to identify those developers who made significant changes to specific code units; in other words, those developers who can be viewed as authors of the code and therefore have expertise on it. However, software systems are in constant change and evolution [Lehman et al., 1997]. For this reason, source code authorship is fundamentally different from authorship in other contexts, like in books or scientific research papers, where the authors are explicitly informed and do not change with time. In software, code artifacts are created by one developer, but later changed by possibly hundreds of developers [Fritz et al., 2014]. On the one hand, this dynamic behavior makes it particularly challenging to identify the authorship of code elements. On the other hand, it only increases the importance of having updated authorship information for software system components. Finally, in distributed development environments, as usual nowadays, identifying authorship is especially a relevant task. In such cases, the geographic distance and consequent communication problems can lead to poor interactions among developers, making more complex to infer who is responsible or expert in each part of the system [Herbsleb et al.,

2001; Mistrík et al., 2010].

In this scenario, version control systems (VCS) such as Subversion (SVN) or Git provide key data to identify code authorship, because they keep information about the changes the developers made in the code. Taking advantage of this source of data, several techniques were proposed to infer and measure code authorship [McDonald and Ackerman, 2000; Mockus and Herbsleb, 2002a; Girba et al., 2005; Bird et al., 2011; Casalnuovo et al., 2015; Rahman and Devanbu, 2011; Fritz et al., 2014]. In summary, these techniques were proposed to recommend code experts [Minto and Murphy, 2007; Begel et al., 2010], to notify developers on changes of interest [Hattori and Lanza, 2009; Ma et al., 2009] and to reveal the relationship between code authorship and software quality [Bird et al., 2011; Rahman and Devanbu, 2011; Greiler et al., 2015; Thongtanunam et al., 2016]. However, *the usage of code authorship to investigate how the implementation work is organized in modern software systems is not widely explored*. We argue that using authorship metrics to investigate development teams can provide interesting insights about the developers who indeed drive the development of software projects, avoiding potential biases resulting from counting a large number of minor contributors. Additionally, code authorship has an inherent potential to address important software engineering problems, such as to assess concentration of knowledge in software projects. This potential can be explored by proposing and evaluating new practical applications of code authorship.

1.2 Proposed Thesis

In this thesis, we propose to answer four overarching questions related to *measuring* and *using* code authorship in software projects. We start by investigating code authorship parameters in 115 popular GitHub projects, including an in-depth analysis of how these parameters evolve in the Linux kernel (**Q1**). After that, by relying on code authorship techniques to identify key developers, we propose a new approach to estimate truck factors—a popular metric that measures concentration of knowledge in software projects (**Q2**). Then, we investigate how common truck factor are events in open source systems (**Q3**). Finally, in the last question, we compare and investigate limitations of existing code authorship techniques to identify software maintainers (**Q4**). More details and motivations for each question are described next.

Q1. How is authorship organized and how does it evolve in software systems?

Recent studies [Meneely and Williams, 2009; Bird et al., 2011; Lavallée and Robillard, 2015; Thongtanunam et al., 2016] confirm that human factors play a significant role in the quality of software components. Other studies [Nagappan et al., 2008; Bird et al., 2008; Cataldo et al., 2012] investigate to what extent the social organization of development teams impact the system implementation. Essentially, these studies confirm the importance of managing the social aspects of systems’ development. However, controlling social factors and promoting organization patterns is a challenging task, which grows exponentially in distributed development environments [Mockus and Herbsleb, 2001; Aspray et al., 2006; Jiménez et al., 2009; Prikladnicki et al., 2010]. In many cases, the development team organization is not previously designed or planned, it just arises during the development process. We argue this problem can be mitigated by setting to build an empirical body of knowledge on how authorship-related measures evolve in successful systems. Although previous studies investigate the organization of development teams [Godfrey and Qiang Tu, 2000; Mockus et al., 2002; Koch and Schneider, 2002; Dinh-Trong and Bieman, 2005], they do not rely—to the best of our knowledge—on authorship metrics to identify the main contributors when analyzing the work force of a system. Notwithstanding, ignoring the relative importance of these developers can lead to misleading conclusions. The reason is that systems may have thousands of developers, but usually, only a small portion of them drive the real development [Mockus et al., 2002; Goeminne and Mens, 2011; Joblin et al., 2017]. In other words, it is common that most developers perform only minor contributions, particularly in open source projects. Therefore, in this first question *our goal is to rely on authorship measures to identify the key developers of a system, in order to provide insights about the developers that indeed conduct a system development*. This investigation aims to provide general information about the authorship organization, such as distribution of files per author, developers profile, and collaboration patterns.

Q2. Can code authorship metrics be used to estimate truck factors?

In the previous question (**Q1**), we propose to investigate how code authorship is organized and evolves in a large number of systems. In this second question, we investigate the usage of code authorship in a more specific scenario: to assess concentration of knowledge in software projects. To conduct this investigation, we adopt a well-known concept proposed by the agile community, called truck factor (TF). A system’s truck factor is defined as “*the number of people on your team that have to be hit by a truck (or quit) before the project is in serious trouble*” [Williams and Kessler, 2003]. Systems with a low truck factor spot strong dependencies towards a small set of developers, probably suggesting the existence of knowledge silos in development teams. If such

important developers abandon the project, the system’s maintenance can be seriously compromised, leading to delays in launching new releases, and ultimately to the discontinuation of the project as whole. To prevent such issues, it is important to have metrics to compute truck factors, which can contribute to reveal knowledge concentration problems in project systems.

Although widely discussed among *eXtreme Programming* (XP) practitioners, there are few studies providing and validating truck factor measures for a large number of systems. In part, the absence of these studies is due to scalability problems of existing approaches [Ricca et al., 2011; Hannebauer and Gruhn, 2014]. Their applicability is usually limited to systems with small number of developers. Essentially, these approaches do not rely on authorship metrics to identify important developers, but instead consider that all developers who changed a file have knowledge on it. Therefore, in this second question, *our goal is to investigate whether authorship metrics can be used to estimate truck factors in real systems*, by focusing the analysis in the most important developers instead of considering the entire set.

Q3. How common are truck factor events in open source projects?

In the previous question (Q2), we investigate the use of code authorship metrics to estimate truck factors. However, there is still a lack of studies that go beyond measuring TF and investigate how common TF events are, i.e., situations where all TF developers abandon the project. Essentially, in order to represent a real risk to software development, TF events should occur with some frequency in the development history of a large sample of systems. Additionally, by investigating how developers handle such events, we can get important insights on how to overcome them. In particular, open source projects represent an interesting case of study. On the one hand, they are more susceptible to lose important developers because they usually have no financial support and, in many cases, the developer’s involvement with the project is limited because this activity does not represent his/her main job [Eghbal, 2016; Coelho and Valente, 2017]. On the other hand, open source projects usually have an active community of developers and users, which can provide new skilled contributors, if needed [Jensen and Scacchi, 2007]. Therefore, in this third question *we investigate how common TF events are in open source systems and how these systems handle such events*.

Q4. Can code authorship identify software maintainers?

Changes made by developers without the adequate expertise can cause a degradation of a system’s structure, which makes expensive to update the software, a problem known as “software aging” [Parnas, 1994]. To attenuate this problem, when a piece of code

needs to be changed, it is important to identify who the skilled developers on it are . However, this is a challenging task, specially in distributed and collective development environments, as usual nowadays. In this context, code authorship metrics can help to identify skilled software developers. Although previous studies have proposed the use of code authorship metrics to identify experts [McDonald and Ackerman, 2000; Mockus and Herbsleb, 2002b; Fritz et al., 2014], the effectiveness of the existing techniques, as well as their possible limitations, remain unclear. Therefore, in this last question, *we assess the use of code authorship techniques to identify skilled developers for specific code units.*

1.3 Outline

The studies that comprise the core of this thesis, with the exception of one, were published in software engineering conference and journals. Therefore, the thesis' chapters preserve the original structure of the manuscripts in order to facilitate independent read. Due to this structure, although all chapters have their particular contributions, some redundancy can be found in the procedures and methodologies. We organized the remainder of this work as follows:

Chapter 2: Measuring and Analyzing Code Authorship. In this chapter, we initially present a detailed case study with the Linux kernel, analyzing aspects of its development organization using code authorship measures, therefore addressing **Q1**. Our analysis accounts for 56 stable releases, spanning a period of over 11 years of development. In summary, this analysis reveals that *(i)* the distribution of the number of files per author in the Linux kernel is highly skewed, *(ii)* most authors are specialists, and *(iii)* authors with a high number of co-authorship connections tend to work with authors with fewer connections. Additionally, we contrast these results with the ones of an extended dataset, composed of 114 projects, and we show that most of the authorship patterns observed in the Linux kernel are also common in other open source systems. This chapter consists of the following publication:

Avelino, G., Passos, L., Hora, A., and Valente, M. T. (2017) Assessing Code Authorship: The Case of the Linux Kernel. In *13th International Conference on Open Source Systems (OSS)*, pages 151-163.

Chapter 3: Estimating Truck Factors. In this chapter, we investigate a practical application of code authorship, by proposing a new approach to estimate truck factors, therefore addressing **Q2**. We apply the proposed approach in a corpus of 133 popu-

lar open source projects hosted on GitHub. The results indicate that the majority of these projects have a low truck factor ($TF \leq 2$). Additionally, we provide empirical evidence of the reliability of our truck factor estimates, as a product of surveying the main contributors of the target systems. From the survey, we also report the practices that developers argue as most useful to overcome a truck factor event. This chapter consists of the following publication:

Avelino, G., Passos, L., Hora, A. C., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1-10.

Chapter 4: Investigating Truck Factor Events. In this chapter, we address **Q3** by investigating truck factor events in a dataset built with 1,932 popular GitHub projects. We identify that 315 projects (16%) experienced at least one truck factor event and among them, 128 projects (41%) survived their most recent observed truck factor event. We conclude the study by presenting the results of a survey with 33 developers that helped the surviving systems to overcome the detected TF events.

Chapter 5: Identifying Software Maintainers. In this chapter, we selected three code authorship techniques and evaluated their effectiveness on supporting software maintainers recommendation, addressing **Q4**. The selected metrics are: (1) the Number of Commits [Bird et al., 2011; Casalnuovo et al., 2015]; (2) the Number of Lines of Code in the Last Version [Girba et al., 2005; Rahman and Devanbu, 2011]; and (3) the Degree of Authorship (DOA) [Fritz et al., 2014]—a linear regression approach for defining experts. We apply the three techniques in 10 projects (2 commercial and 8 open-source) and compare the results with an oracle we built from surveying 159 developers. Additionally, we investigate the cases where the three techniques fail. We also suggest opportunities of improvements by controlling information about file size and recency of the changes. This chapter consists of the following publication:

Avelino, G., Passos, L., Petrillo, F., and Valente, M. T.(2018). Who Can Maintain this Code? Assessing the Effectiveness of Repository-Mining Techniques for Identifying Software Maintainers. In *IEEE Software*, pages 1-15.

Chapter 6: Conclusion. This final chapter concludes the thesis and gives suggestions for future research.

Chapter 2

Measuring and Analyzing Code Authorship

Code authorship is a key information about large-scale software projects. Among others, it reveals division of work, key collaborators, and developers' profiles. Seeking to better understand authorship in large and successful open source communities, we take the Linux kernel as our first case study. In total, we analyze authorship across 56 stable releases. Our analysis is centered around the Degree-of-Authorship (DOA) metric, which accounts for first authorship events (file creation), as well as further code changes. Authorship along the Linux kernel evolution reveals that: (i) only a small portion of developers (26%) makes significant contributions to the code base; this ratio is almost constant during the Linux kernel evolution; (ii) the number of files per author is highly skewed—a small group of top-authors (3%) is responsible for hundreds of files, while most authors (75%) are responsible for at most 11 files; (iii) most authors in Linux (75%) are specialists and the relation between specialists and generalists tends to be constant; (iv) authors with a high number of co-authorship connections tend to work with authors with fewer connections. Furthermore, we replicate the study in an extended dataset, composed of 114 projects. We identify that most of the authorship patterns observed in the Linux kernel are also common to other open source systems.

2.1 Introduction

Software engineering is essentially a collective effort, requiring the coordination of large developer teams [Crowston and Howison, 2005; Herbsleb, 2007; Mistrík et al., 2010]. To tackle complexity and size, software systems are usually partitioned into subsystems, allowing developers to parallelize implementation [Parnas, 1972]. Hence, collaborative

work and modularization are key players in software development, specially in the context of open source systems.

In a collaborative setup imposed by open source development, code authorship allows maintainers to assess overall division of work among project members (e.g., to seek better working balance), identify profiles within the team (e.g., specialists versus generalists), and find best fitting developers for a target task.

In this chapter, we initially set to understand authorship in a large and long-lived successful system—the Linux kernel. Our goal is to identify authorship parameters from the Linux kernel evolution history, as well as interpret why they appear as such. At all times, we also check whether those parameters apply to the subsystem level, allowing us to assess their generality across different parts of the kernel. This analysis accounts for 56 stable releases (v2.6.12–v4.7), spanning a period of over 11 years of development (June, 2005–July, 2016). Additionally, in a second study, we contrast the authorship results computed for the Linux kernel with the ones computed for a dataset of 114 popular open source systems, retrieved from GitHub.

First Study (Linux Kernel). First, when investigating the Linux kernel authorship history, we provide answers to four research questions:

RQ1. What is the proportion of developers ranked as authors?

Motivation: In large open source communities, most developers perform occasional and minor contributions [Mockus et al., 2002; Goeminne and Mens, 2011; Joblin et al., 2017]. With that insight, not all contributors perform significant changes, but how many do? Hence, this research question allows to reveal the proportion of developers with significant contributions to Linux development, defining the project working force to be studied.

RQ2. What is the distribution of the number of files per author?

Motivation: Answering such a question provides us with a measure of the work overload within team members, as well as how that evolves over time.

RQ3. How specialized is the work of Linux authors?

Motivation: Following the Linux kernel architectural decomposition, we seek to understand the proportion of developers who have a narrower understanding of the system (specialists), versus those with a broader knowledge (generalists). Specialist developers author files in a single subsystem; generalists, in turn, author files in different subsystems. Answering this research question seeks to assess how effective the Linux kernel

architectural decomposition is in fostering specialized work, a benefit usually expected from a good modularization [Sullivan et al., 2001; Baldwin and Clark, 1999].

RQ4. What are the properties of the Linux co-authorship network?

Motivation. The authorship metric we use enables identifying multiple authors per file, evidencing a co-authorship relation among developers [Meneely and Williams, 2011]. Such relations form a network—vertices denote authors and edges connect authors sharing common authored files. This question seeks to identify co-authorship properties in the Linux kernel evolution. We compute and discuss several properties, including mean degree, number of solitary vertices, clustering, and assortative coefficients, among others.

Second Study (114 open source projects). We extended the initial study (Linux kernel) by applying the same authorship-related metrics to a dataset of 114 popular (number of stars) open source systems, retrieved from GitHub. We contrast the studies results and observe that most of the authorship patterns firstly identified in the Linux kernel are also present in this extended dataset. For example, most projects in the extended dataset present skewed distribution of the number of files per author and a high number of specialists.

Organization. The remainder of this chapter is organized as follows. Section 2.2 provides a description of our study design. Section 2.3 details our results, providing answers for the four research questions. Section 2.4 discusses our key findings when investigating these questions. Section 2.5 compares the Linux kernel measures with the ones computed for other 114 open source projects. Sections 2.6 and 2.7 discuss threats to validity and related work, respectively. Section 2.8 concludes the chapter.

2.2 Study Design

In this section, we first present the metric we used to identify source code authors. Then, we describe the Linux kernel architectural decomposition and how we collect and process the development data used in this study.

2.2.1 Author Identification

At the core of our study lies the ability to identify and quantify authorship at the source code level. To identify file authors, as required by our five research questions, we employ a normalized version of the *degree-of-authorship* (DOA) metric [Fritz et al.,

2010, 2014]. The metric is originally defined in absolute terms:

$$DOA_A(d, f) = 3.293 + 1.098 * FA + 0.164 * DL - 0.321 * \ln(1 + AC) \quad (2.1)$$

From the provided formula, the absolute degree of authorship of a developer d in a file f depends on three factors: first authorship (FA), number of deliveries (DL), and number of acceptances (AC). If d is the creator of f , FA is 1; otherwise it is 0; DL is the number of changes in f made by d ; and AC is the number of changes in f made by other developers. DOA_A assumes that FA is by far the strongest predictor of file authorship. Further changes by d (DL) also contribute positively to his authorship, but with less importance. Finally, changes by other developers (AC) contribute to decrease someone’s DOA_A , but at a slower rate. The weights in Equation 2.1 stem from an experiment with professional Java developers [Fritz et al., 2014]. We reuse such thresholds without further modification.

Then, we define the normalized DOA (DOA_N):

$$DOA_N(d, f) = DOA_A(d, f) / \max(\{DOA_A(d', f) \mid d' \in \text{changed}(f)\}) \quad (2.2)$$

In Equation 2.2, $\text{changed}(f)$ denotes the set of developers who edited a file f up to a snapshot of interest (e.g., release). This includes the developer who creates f , as well as all those who later modify the file. $DOA_N \in [0..1]$: 1 is granted to the developer with the highest absolute DOA among those changing f ; in any other case, DOA_N is less than one.

Lastly, the set of authors of a file f is given by:

$$\text{authors}(f) = \{d \mid d \in \text{changed}(f) \wedge DOA_N(d, f) > 0.75 \wedge DOA_A(d, f) \geq 3.293\} \quad (2.3)$$

The authors identification (Equation 2.3) depends on specific thresholds— 0.75 and 3.293. Those stem from a calibration setup when applying this equation to a large corpus of open-source systems. Details about the calibration study are in Section 3.4.2.

DOA Trade-offs. Identifying code authorship is not trivial. Due to constant changes by different collaborators, authorship is not granted to a single developer. Rather, a single file may have different authors, with varying groups as the system evolves. Using DOA certainly has limitations. For instance, DOA does not account for the number of lines developers change on a given commit, nor why such lines are changed. On the other side, the metric does have its strengths. In addition to file creation, DOA accounts for different authorship events (e.g., deliveries and acceptances). The metric

Table 2.1. Linux subsystems (release v4.7)

Subsystem	# Files	%	
Driver	22,943	42%	████████
Arch	17,069	32%	██████
Misc	6,621	12%	██
Core	3,840	7%	█
Net	1,957	4%	█
Fs	1,809	3%	█
Firmware	151	0%	
Total	54,400	100%	

is also language-agnostic, as it does not consider file content, which favors automatic mining of software repositories.

2.2.2 Linux Kernel Architectural Decomposition

Investigating authorship at the subsystem level requires a reference architecture of the Linux kernel, as well as a mapping between elements at the source code level to elements in the architectural model. Structurally, the Linux kernel architectural decomposition comprises seven major subsystems [Corbet et al., 2005]: *Arch* (architecture dependent code), *Core* (scheduler, IPC, memory management, etc), *Driver* (device drivers), *Firmware* (firmware required by device drivers), *Fs* (file systems), *Net* (network stack implementation), and *Misc* (miscellaneous files, including documentation, samples, scripts, etc).

To map files in each subsystem, we rely on expert knowledge. Specifically, we use the mapping rules set by G. Kroah-Hartman, one of the main Linux kernel developers.¹ Table 2.1 provides information about the size, as measured by number of files, in each kernel subsystem. As the table shows, *Driver* is the largest subsystem, followed by *Arch*, *Misc*, and *Core*.

2.2.3 Data Collection

We study 56 stable releases of the Linux kernel, obtained from `linus/torvalds` GitHub repository.² A stable release is any named tag snapshot whose identifier does not have a `-rc` suffix. To define the *authors* set of a file f in a given release r , we calculate DOA_N from the first commit up to r . Hence, all files from each release have at least

¹<https://github.com/gregkh/kernel-history/blob/master/scripts/stats.pl>

²<https://github.com/torvalds/linux>

one author. It happens, however, that the Linux kernel history is not fully stored under Git. Linus Torvalds explains:³

I'm not bothering with the full history, even though we have it. We can create a separate "historical" git archive of that later if we want to, and in the meantime it's about 3.2GB when imported into git - space that would just make the early git days unnecessarily complicated, when we don't have a lot of good infrastructure for it.

We use `git graft`⁴ to join the history of all releases prior to v2.6.12 (the first release recorded in Git) with those already controlled by Git (\geq v2.6.12). After joining, we increment the Linux kernel Git history with 64,468 additional commits.

Given the entire Linux kernel evolution history, we then query the git repository to filter stable release names, checking out each stable release at a time. For a given release snapshot, we list its files, calculating the set of authors (Equation 2.3) for each of them. In the latter case, we rely on `git log --no-merges` to discard merges and retrieve all the changes to a given file prior to the release under investigation.

It is worth noting that prior to calculating the files authors, we map possible aliases among developers, as well as eliminate unrelated source code files. Next, we detail such steps.

Alias Detection. One core challenge in mining software repositories consists of alias detection. An alias occurs when a single developer uses different identities to commit changes; in such cases, one should track all changes as being of the same developer. In Git, an identity comprises a username and email. To track changes in the face of aliases, we first assign to a single developer all commits with exactly the same e-mail, but with different names. For example, the email `dmonakhov@openvz.org` associates to six different names: “Dmitry Monakhov”, “Monakhov Dmitriy”, “Dmitri Monakhov”, “Dmitri Monakho”, “Dmitry”, and “Dmitriy Monakhov”. Then, we consider developer names to be the same if they have a Levenshtein distance [Navarro, 2001] of at most one. Such distance corresponds to the minimal number of single-character insertions, deletions, or substitutions required to make two strings identical, respected the limit of a single-character change. For example, “Haavard Skinnemoen” and “Havard Skinnemoen” are considered to be the same name, since one insertion is needed to make the two strings equal.

³<https://github.com/torvalds/linux/commit/1da177e4c3f41524e886b7f1b8a0c1fc7321cac2>

⁴<https://git.wiki.kernel.org/index.php/GraftPoint>

File Cleaning. Code authorship should consider only the files representing the source code of a target system. Thus, it should ignore documentation, images, examples, third-party source code files, etc. To filter out files unrelated to the Linux kernel, we use the Linguist library.⁵ GitHub uses the latter to identify a system’s language, as well as files that should not be counted as part of the system (e.g., when collecting repository statistics). We use Linguist to exclude unrelated files at each release we analyze. For example, release v4.7 contains 54,400 files; of those, we remove around 20%. Most of the files we exclude consist of documentation (4,560) and device-tree specifications (2,013).⁶ Additionally, we remove the *Firmware* subsystem from further analysis, as most of its files are blobs.

2.2.4 Custom-made Infrastructure

Using custom-made scripts, we fully automate authorship identification, as well as the collection of supporting data for the claims we make. Our infrastructure is publicly available on GitHub.⁷ We encourage others to use it as means to independently replicate/validate our results.

2.3 Results

In this section, we present the results we obtain by investigating the four research questions introduced in Section 2.1

RQ1. Proportion of Authors

What is the proportion of developers ranked as authors?

In the latest release in our analysis (v4.7), the Linux kernel has 13,436 developers; of those, 26% author at least one file. Figure 2.1a contrasts the number of developers with the number of authors across different releases. In both cases, we identify a steady increase, with similar growth rates. Specifically, there is an 8-fold increase in the number of authors, from 432 (first release) to 3,459 (last release). The number of developers, in turn, has grown 8.5 times.

Historically, the mean proportion of authors is 26.8%, with alternating periods of small increases and decreases—see Figure 2.1b. Throughout the kernel development,

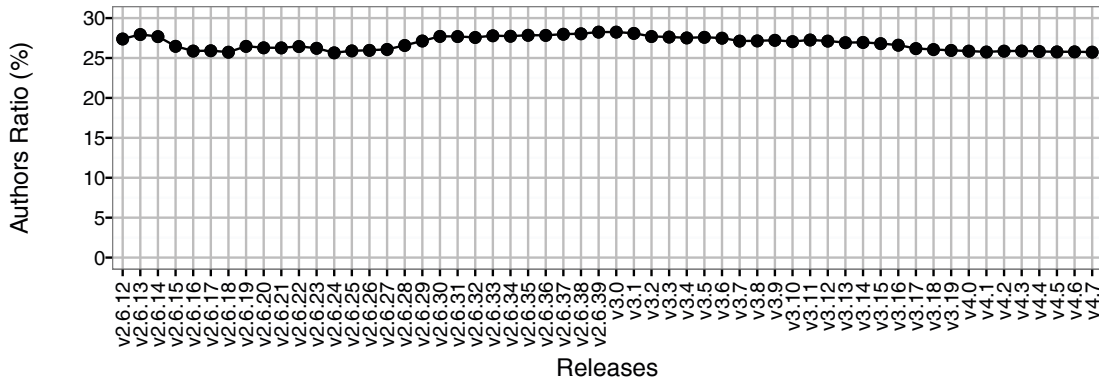
⁵<https://github.com/github/linguist>

⁶Files ending in `dts` and `dtsi`.

⁷https://github.com/gavelino/data_oss17



(a) Number of authors and developers over time



(b) Proportion of authors over time

Figure 2.1. Authors and developers over time

Table 2.2. Author Ratio Evolution: Summary Statistics. Avg: Average, Std Dev: Standard Deviation

Subsystem	Min	Max	Avg \pm Std Dev
Core	22.80%	28.39%	25.77 \pm 1.56%
Driver	23.82%	26.87%	25.00 \pm 0.80%
Arch	30.00%	35.32%	33.10 \pm 1.28%
Net	12.28%	15.84%	13.63 \pm 0.90%
Fs	9.85%	17.26%	12.61 \pm 1.95%
Misc	11.88%	22.73%	14.85 \pm 2.69%
All	25.66%	28.26%	26.86 \pm 0.83%

the proportion of authors is nearly constant ($Std\ dev = \pm 0.83\%$). Thus, the heavy-load maintenance of the kernel has been kept in the hands of a little more than one quarter of all developers.

Contrasting the authorship proportion at the subsystem level with the global one shows that *Core* (mean 26%) and *Driver* (mean 25%) approximate to the global average.⁸ As Table 2.2 shows, both subsystems display little variance in authorship ratio, which follows directly from their low standard deviation (Std Dev). To a lesser extent, the authorship ratio in *Arch* (33%) also approximates the global parameter; the same does not occur for *Net*, *Fs*, and *Misc*. We interpret such discrepancies as follows.

As *Core* and *Arch* provide the basic functionality for the remaining parts of the system, developers must have great confidence on the changes they propose, discouraging volunteers from performing small changes as a means to become kernel contributors. In the latter case, authorship ratio increases. In addition, changing *Arch* requires vendor-specific expertise when maintaining support for different CPUs. Thus, knowledge becomes narrower, lifting author ratio. *Driver* follows a similar rationale, requiring manufacturer-specific knowledge about the hardware to support.

Different from the latter three, *Net* and *Fs* require less hardware-specific knowledge; in a sense, maintaining such subsystems is somehow easier, making them more attractive for occasional and minor contributions. As a consequence, there is a decrease in authorship.

Misc, due to a mix content, fits different goals, not necessarily in-tune with the operating system itself (e.g., infrastructure for building the kernel). Its size, as given in lines of code, tends to be stable across the kernel, indicating that *Misc* does not change frequently [Passos et al., 2015].

From such insights, we hypothesize that the closer a subsystem is to vendor-specific code (e.g, *Arch* and *Driver*) or to the "brain" of the kernel (*Core*), the harder is to maintain code, inflating authorship.

In the last release, 26% of the Linux's developers are authors, concentrating the heavy-load of the Linux kernel maintenance. Such proportion is almost constant over time. The proportion of authors is higher in subsystem that requires vendor-specific knowledge (e.g., *Driver* and *Arch*) or an understanding of the kernel as a whole (e.g., *Core*).

RQ2. Distribution of the Number of Files per Author

What is the distribution of the number of files per author?

⁸We use the terms *average* and *mean* interchangeably. Both should be interpreted as the arithmetic mean.

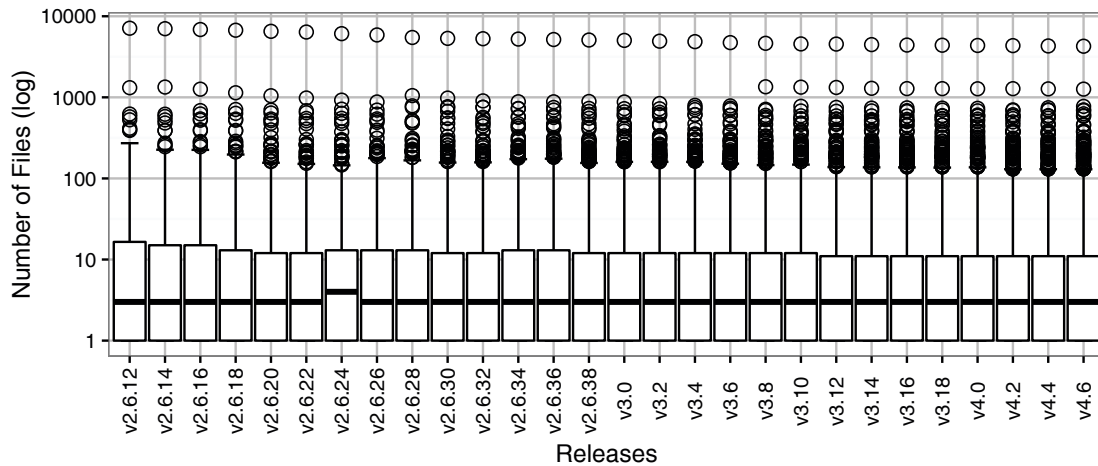


Figure 2.2. Distribution of the number of files per author in each release

The number of files per author is highly skewed. Figure 2.2 presents the boxplots of files per author across all 56 releases (we adjust the boxplots for skewness—see [Hubert and Vandervieren, 2008]). To simplify the visualization of the results, we present the boxplots at each two releases. Globally, 50% of the authors responds to at most four files (median), a measure that remains constant across all releases except one (v2.6.24); for 75% of the authors (third quartile), the max number of files per author ranges from 11 to 16 along the releases. Outliers follow from the skewed distribution. Still, the number of authors with more than 100 files is always lower than 7% of the authors, ranging from 7% in the first release to 3% in the last one.

It is interesting to note that file authorship follows a pyramid-like shape of increasing authority; at the top, Linus Torvalds acts as a "dictator", centralizing authorship of most of the files (after all, he did create the kernel!). Bellow him lies his hand-picked "lieutenants", often chosen on the basis of merit. Such organization directly reflects the Linux kernel contribution dynamics, which is itself a pyramid [Bettenburg et al., 2015]. However, as the kernel evolves, we see that Torvalds is becoming more "benevolent". As Figure 2.3 shows, the percentage of the Linux kernel files authored by Torvalds has reduced from 45% (first release) to 10%, in v4.7. This decrease is not a simple result of the Linux kernel growth. Actually, the number of files authored by him decreased 40% in this period, suggesting that many of his files started to be maintained by other developers. Currently, Torvalds spends more time verifying and integrating patches than writing code [Corbet et al., 2013].⁹ His "benevolence" appears to have a direct impact

⁹These results are also consistent with a recent interview from Linus Torvalds, where he acknowledges that nowadays, Linux maintenance has much less dependence on him than 15 years ago. See <http://www.bloomberg.com/news/articles/2015-06-16/>

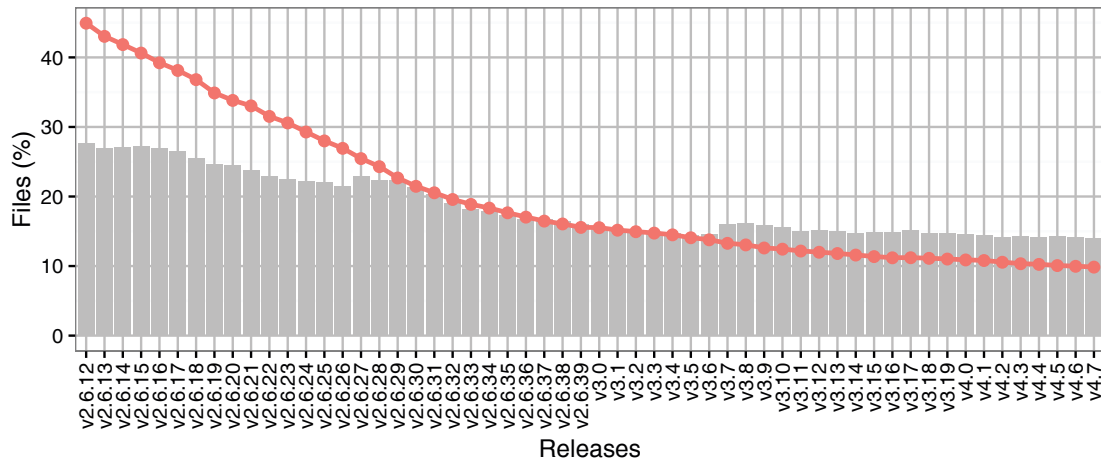


Figure 2.3. Percentage of files authored by the top-10 authors over time. The line represents Linus Torvalds (top-1) and the bars represent the accumulated number of files of the next top-9 authors

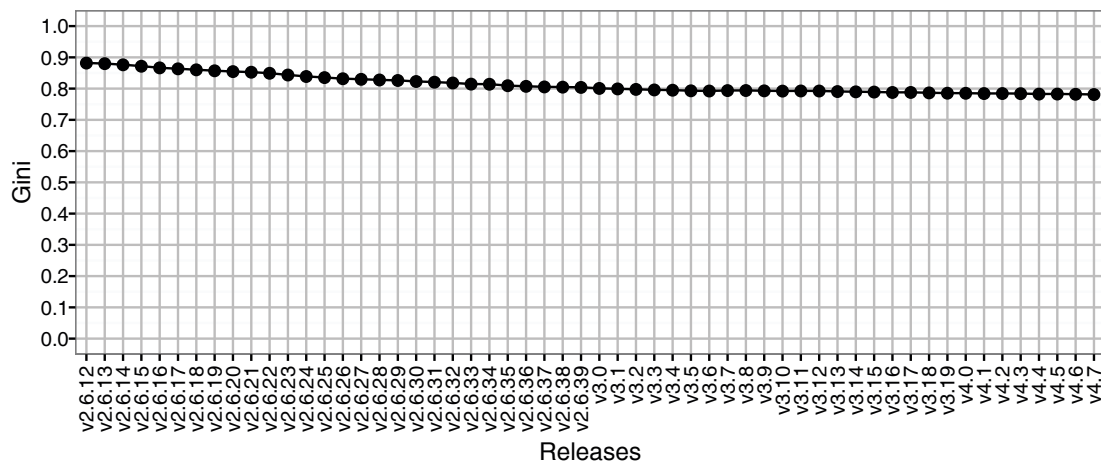


Figure 2.4. Gini coefficients

downwards the authorship pyramid. For instance, the figure shows the percentage of files in the hand of the top-10 Linux kernel authors is consistently decreasing. This suggests that authorship is increasing at lower levels of the pyramid, becoming more decentralized. This is indeed expected and, to an extent, required to allow the Linux kernel to evolve at the pace it does.

To better comprehend the distribution of the number of files per author, we also analyze Gini coefficients (Figure 2.4). Gini is a widely used metric to express the wealth inequality among a target population [Gini, 1921]. Wealth, in this case, stands

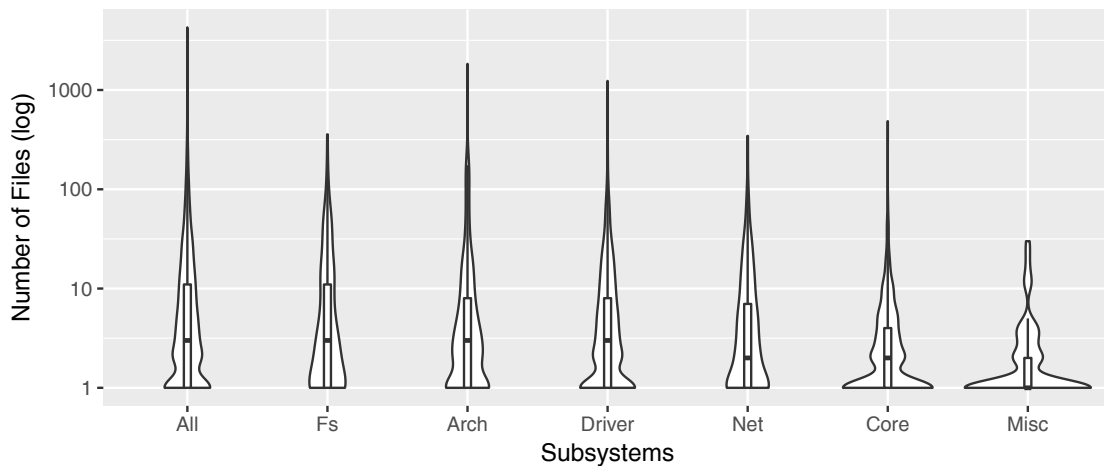


Figure 2.5. Number of files per author (release v4.7)

for the number of files per author. The coefficient ranges from 0 (perfect equality, when everyone has exactly the same wealth) to 1 (perfect inequality, when a single person concentrates all the wealth). In all releases, the Gini coefficient is high, confirming skewness. However, we notice a decreasing trend, ranging from a Gini of 0.88 in the first release to 0.78 (v4.7). Such a trend further strengthens our notion that authorship in the Linux kernel is becoming less centralized. Alternatively, it means that the number of authors per file is becoming more equal.

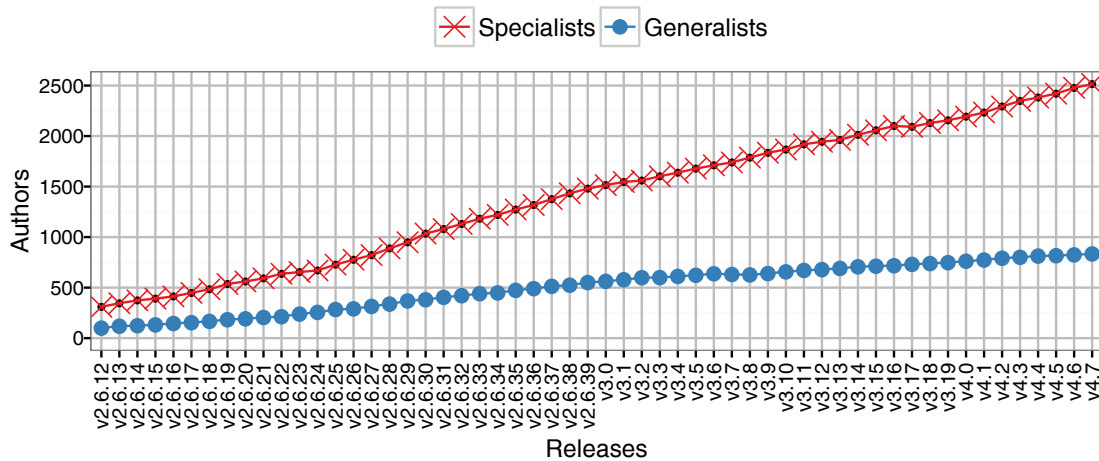
The distribution of files per author is also highly skewed at the subsystem level. For instance, in the last release (see Figure 2.5), the number of files per author up to the 75% percentile in *Fs*, *Arch*, and *Driver* closely resemble one-another and the global distribution as a whole—all share the same median (three). *Core* and *Misc*, however, have less variability than the other subsystems, as well as lower median values (two and one, respectively).

The number of files per author follows a highly skewed distribution in all analyzed releases. However, it is becoming more equal over time. For example, the top-1 author owns 45% of the files in the first analyzed release, but only 10% of the files in the last one.

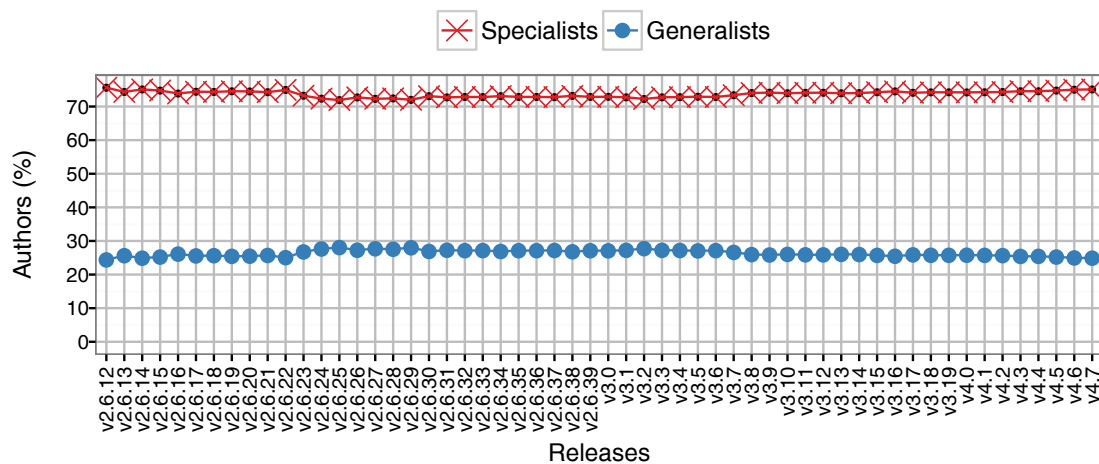
RQ3. Work Specialization

How specialized is the work of Linux authors?

To assess work specialization, we introduce two author profiles. We call authors *spe-*



(a) Absolute number of specialists and generalists



(b) Percentage of specialists and generalists

Figure 2.6. Specialists and generalists over time

specialists if they author files in a single subsystem. *Generalists*, in turn, author files in at least two subsystems. When classifying authors, we ignore files in the `include/` directory. This directory may lead to misclassifications because it was designed to store all Linux kernel headers files, independently of what subsystem the file belongs. As Figure 2.6 shows, the number of specialists dominates the amount of generalists, both in absolute terms as proportionally.

Proportionally, any given release has at least 72% of specialist authors, with a maximum of 76%; at all times, no more than 28% of the authors are generalists. Moreover, the proportion of generalists and specialists appears to be fairly stable across the entire kernel (All) and its constituent subsystems (except for *Misc*)—see Figure 2.7. It is important to note, that as a generalist developer authored files in at least two

different subsystems, the number of generalists shown in All is not the sum of generalists in each subsystem.

Looking at the division of generalists and specialists working in each subsystem also allows to assess how much the Linux kernel architectural decomposition fosters specialized work—a desired by-product of a good modularization design [Sullivan et al., 2001; Baldwin and Clark, 1999]. We notice that the architectural decomposition plays a key role in fostering specialists inside the *Driver* subsystem, but less so elsewhere. Thus, the Linux kernel architecture partially fosters specialization. The reason it occurs so extensively inside *Driver* follows from the plugin interface of the latter and its relative high independence from other subsystems. Device-drivers are supposed to be self-contained modules that are plugged into the kernel and loaded as needed [Corbet et al., 2005; Passos et al., 2015]. There are cases, however, when developers must change other parts of the kernel when adding new device drivers or maintaining them. An example includes scattering code in *Arch* due to hardware detection limitations [Passos et al., 2015]. In cases such as this, the knowledge to perform changes increases, leading to the appearance of generalists within the *Driver* subsystem.

Similar to *Driver*, *Net* and *Fs* also follow a plugin-like model of development. Counter-intuitively, however, both subsystems display a dominance of generalists. Our hypothesis is that the maintenance of these systems interplays with other parts of the kernel. For instance, a new distributive file system may require specific network protocols to be added or fine-tuned. Others also report a similar understanding [Bowman et al., 1999].

The results in *Arch* and *Core* match our expectations. When maintaining either subsystem, developers must be aware of possible side effects elsewhere; also, modularization of both systems is harder to achieve, fostering less specialization. Often times, for instance, developers must break programming interfaces to get better performance. Other cases include the maintenance of CPU-specific code that has been historically troublesome (e.g., ARM).¹⁰ Specifically, *Core* is the subsystem with the lowest percentage of specialized workers (21%). This is also expected since *Core* developers tend to have expertise on Linux’s central features, which allows them to also work on other subsystems.

Fine-grained specialization. We also investigate the specialization at a fine-grained level. By adopting a more generic strategy, we consider each top-level directory as a Linux kernel module.¹¹ By applying this approach, in the last release (v4.7), we

¹⁰<https://lkml.org/lkml/2011/3/17/492>

¹¹We discard files in the root and include directories.

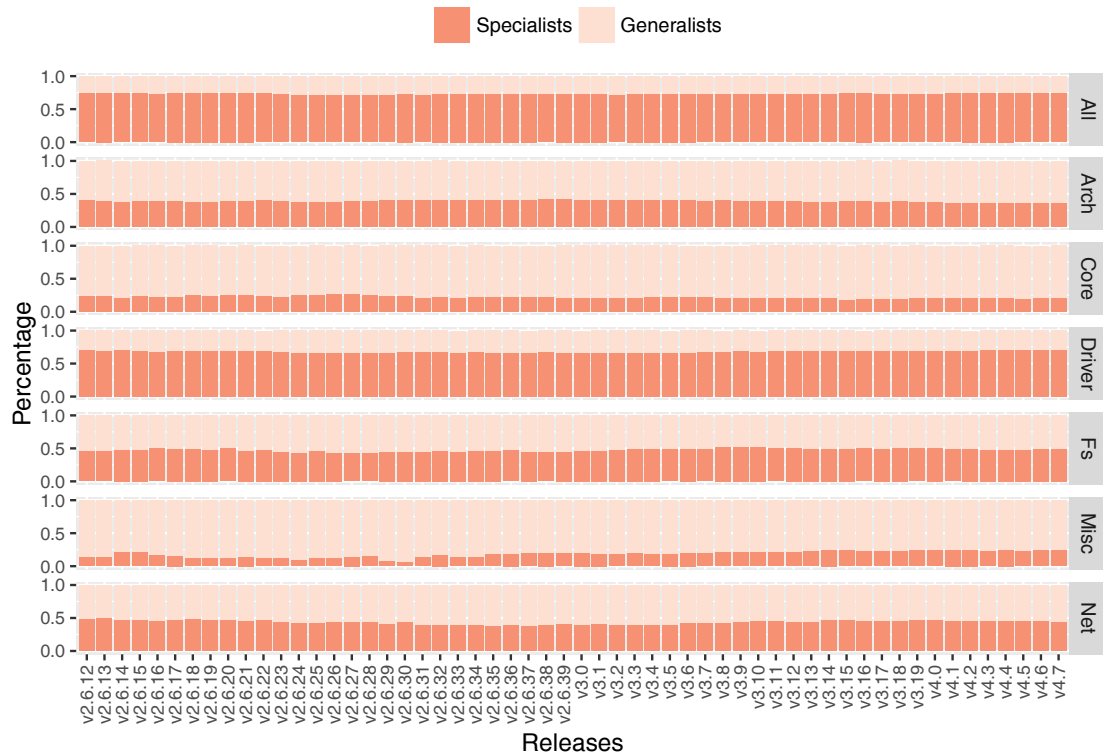


Figure 2.7. Percentage of specialists and generalists

divided the Linux kernel in 16 modules and the percentage of specialists and generalists are, respectively, 73% and 27%. The results at top-level directory are similar to the obtained adopting the kernel subsystems division (respectively, 75% and 25%, in v4.7). In other words, even considering a finer granularity (16 modules, instead of 6 subsystems) the specialization remains high in the Linux kernel.

Specialization is a common practice in the Linux kernel development over the years. In 11 years of development, at least 72% of the Linux authors are specialists, i.e., all files they authored are located in a single subsystem. This is especially valid for Driver authors. The most notable exception are Core developers, who tend to be generalists.

RQ4. Co-authorship Properties

What are the properties of the Linux co-authorship network?

In this section, we investigate the collaborative nature of the Linux implementation

Table 2.3. Percentage of files with multiple authorship (release v4.7)

All	Driver	Arch	Core	Net	Fs	Misc
28%	28%	28%	27%	30%	24%	20%

work. First, it is important to clarify that the author identification model we use allows multiple authors per file; they only need to have a DOA_N value that fits the thresholds defined in Section 2.2.1. We compute the percentage of files with multiple authorship per subsystem, as presented in Table 2.3. As we can see, most files have a single author; however, the percentage of files with multiple authors is relevant. It ranges from 20% of the files in *Misc* to 30% in *Net*. When considering the files in all subsystems, it reaches 28%.

As many files in the Linux kernel result from the work of different authors, we set to investigate such collaboration by means of the properties of the Linux kernel *co-authorship network*. We model the latter as follows: vertices stand for Linux kernel authors; an edge connects two authors v_i and v_j if $\exists f$ such that $\{v_i, v_j\} \subseteq authors(f)$. To strength the collaboration meaning of the co-authorship network, we filter edges without a temporal overlap between the file authorship. In other word, we only connect the developers v_i and v_j if at least one change in f performed by v_i or v_j occur while the other developer is active (by considering her first and last commit in the repository).

Differently from books and scientific papers, our co-authorship network does not account for a possible hierarchy among authors (first author, second author, etc). Rather, we take all the authors of a file to be equally important co-authors.

To answer our research question, we initially analyze the latest co-authorship network of the entire kernel, as given by the last release in our corpus (v4.7). When doing so, we measure four metrics: *number of vertices*, *mean degree*, *clustering coefficient*, and *assortative coefficient*. At all times, we contrast the system level network with those at the subsystem level. Additionally, we investigate how the values we report came to be, analyzing their historical evolution. Figure 2.8 shows a fragment of the latest co-authorship network (note the central role of Linus Torvalds). Table 2.4 presents the values of our target metrics.¹²

The number of vertices (authors) determines the size of a co-authorship network. The mean degree network, in turn, inspects the number of co-authors that a given author connects to. In the system level network for release v4.7 (All), the mean vertex degree is 3.44, i.e., on average, a Linux author collaborates with 3.44 other authors. At the subsystem level, *Driver* forms the largest network (2,604 authors, 75%), whereas

¹²We use the R *igraph* (version 1.0.1) to calculate all measures.

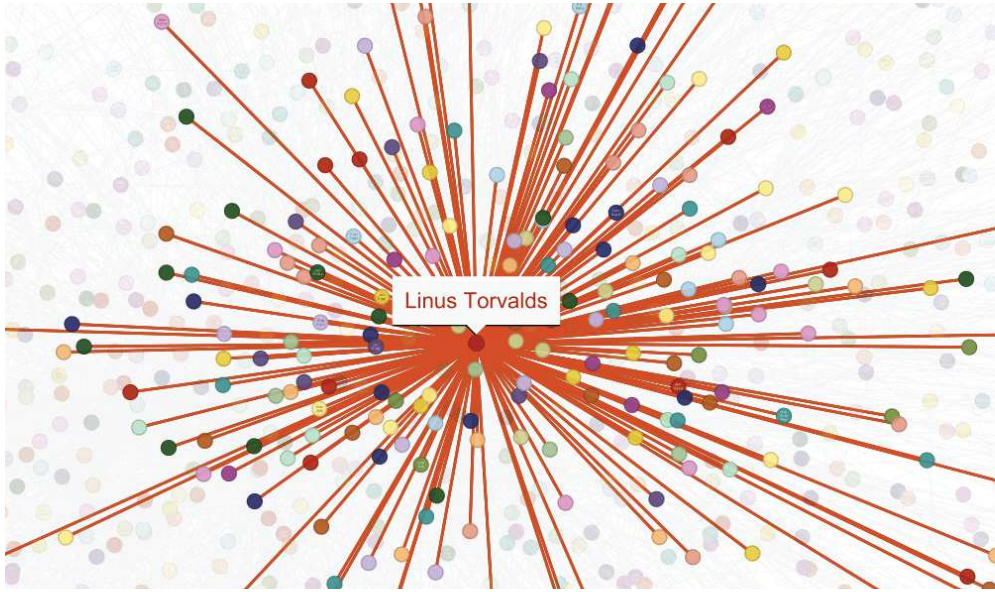


Figure 2.8. Fragment of the Linux co-authorship network

Table 2.4. Co-authorship network properties (release v4.7)

	All	Driver	Arch	Core	Net	Fs	Misc
Vertices	3,459	2,604	1,145	1,083	269	175	78
Mean Degree	3.44	2.56	3.03	1.57	2.46	2.55	0.74
Clustering	0.077	0.069	0.126	0.070	0.199	0.174	0.200
Assortativity	-0.084	-0.130	-0.072	-0.083	-0.025	-0.154	-0.117

Misc results in the smallest one (78 authors, 2%). *Arch* has the highest mean degree (3.03 collaborators per author); *Misc* has the lowest (0.74 collaborators per author). Linus Torvalds has connections with 215 other authors. His collaborations spread over all subsystems and range from 92 collaborations in *Driver* to five collaborations in *Misc*. Excluded Torvalds, the top-2 and top-3 authors with more collaborators have 156 and 117 collaborators, respectively.

The third metric concerns the *clustering coefficient* of the co-authorship network. Also known as graph transitivity, this coefficient reveals the degree to which adjacent vertices of a given vertex tend to be connected [Watts and Strogatz, 1998]. In a co-authorship network, the coefficient gives the probability that two authors who have a co-author in common are also co-authors themselves. A high coefficient indicates that the vertices tend to form high density clusters. The clustering coefficient of the Linux kernel is small (0.077). Nonetheless, *Misc*, *Net*, and *Fs* exhibit a higher tendency to form high density clusters (0.200, 0.199, and 0.174, respectively) in comparison to other subsystems. The three subsystems are the smallest analyzed, a factor that influences

Table 2.5. Number of files authored by Solitary Authors (i.e., authors that do not have co-authors)

Number of Files	1 File	2 Files	3 Files	>3 Files	Total
Solitary Authors	495	182	60	101	838

the development of collaboration clusters [Albert and Barabási, 2002].

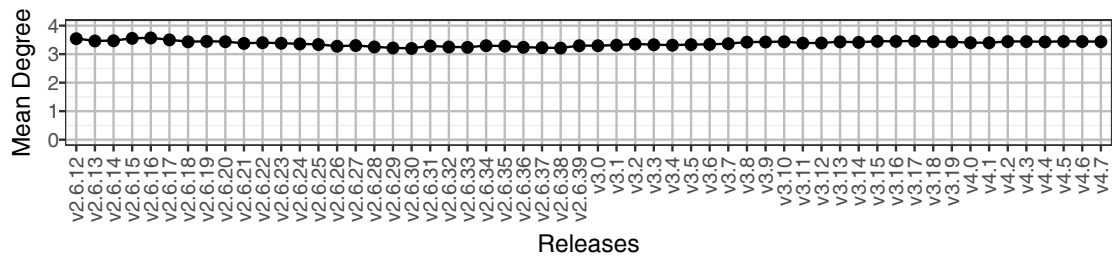
Last, we compute a measure called *assortativity coefficient*, which correlates the number of co-authors of an author (i.e. its vertex degree) with the number of co-authors of the authors it is connected to [Newman, 2003]. Ranging from -1 to 1, the coefficient shows whether authors with many co-authors tend to collaborate with other highly-connected authors (positive correlation). In v4.7, all subsystems have negative assortativity coefficients, ranging from -0.154 in *Fs* to -0.025 in *Net* subsystem. This result diverges from the one commonly observed in scientific communities [Newman, 2004]. Essentially, this suggests that Linux kernel developers often divide work among experts who help less expert ones. These experts (i.e., highly-connected vertices), in turn, usually do not collaborate among themselves (i.e., the networks have negative *assortative coefficients*).

In the co-authorship networks, there is a relevant amount of *solitary authors*—authors that do not have co-authorship with any other developer. As Table 2.5 shows, 24% (838) of Linux kernel developers are solitary, authoring few files. Only 12% of solitary authors have more than three files; a single outlier exists in v4.7, authoring 27 files, all in *Driver*. In fact, it is worth noting that 59% of solitary authors work in the *Driver* subsystem. The latter is likely to follow from the high proportion of specialists within that subsystem (see RQ.3).

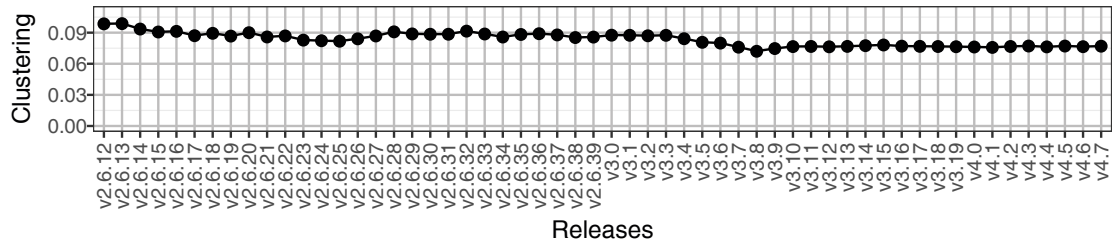
Evolution of Co-authorship network metrics. We set to investigate how the mean degree, clustering coefficient, and assortative coefficients evolved to those in release v4.7. Figure 2.9 displays the corresponding graphics.

Despite a period of decrease, the mean degree (see Figure 2.9a) has little variation from the first release (3.55) to the last one (3.44). Clustering coefficient (see Figure 2.9b), in turn, varies from 0.099 (first release) to 0.077 (v4.7). Since the mean degree does not vary considerably, we interpret such decrease as an effect of the growth of the number of authors (network vertices). The latter creates new opportunities of collaboration, but these new connections do not increase the density of the already existing clusters. A similar behavior is common in other networks, as described by Albert and Barabási [2002].

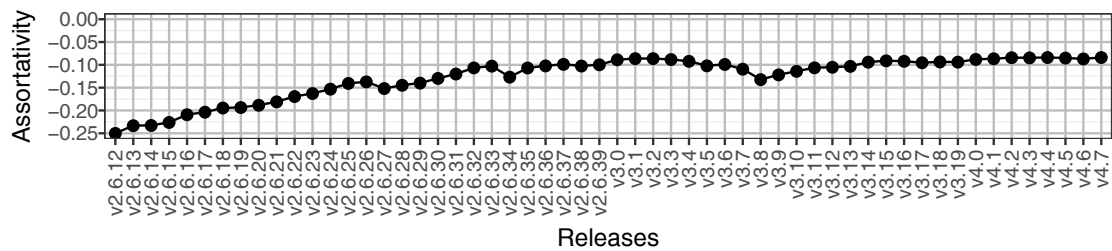
We observe a relevant variation in the evolution of assortativity coefficients—see



(a) Mean degree



(b) Clustering coefficient



(c) Assortativity coefficient

Figure 2.9. Co-authorship network properties over time

Figure 2.9c. Measurements range from -0.255 in the first release to -0.084 in v4.7. Such a trend aligns with the decrease of the percentage of files authored by Linus Torvalds and the other top authors (refer to RQ.3). With less files, these authors are missing some of their connections and becoming more similar (in terms of vertex degree) to their co-authors.

On average, a Linux author collaborates with 3.4 other authors and this number is almost constant over time. Moreover, the co-authorship network indicates that authors that collaborate with many other do so in a way to “help” authors who collaborate less, suggesting some sort of mentorship.

2.4 Discussion

The Importance of Code Authorship. Studies on open source communities can lead to misleading conclusions if the relative importance of the developers is not considered. The reason is that open source systems may have thousands of developers, but they usually have much less authors. In this context, authorship measures—like the DOA model used in this thesis—can help to automatically identify the *key developers* of each file in a large open source project. By collecting authorship data for each file, we can also reason about many organizational aspects of a development team, as reported in this chapter. For example, we used authorship measures to reveal the workload of Linux’s authors (75% of them are authors of at most 11 files) and their degree of specialization (more than 72% of the authors are specialists in a single subsystem). Since the DOA model is computed using commit histories, it can also be used to study the evolution of these measures. For example, after collecting authorship data for 56 Linux releases, we showed that the number of files authored by the top-1 author suffered a major decrease (from 45% of the system’s files to 10%). We also reported that ratio of specialists and generalists is almost constant over time (72% vs 28%).

Interestingly, the Linux Foundation releases an annual report on the state of the kernel development [Corbet et al., 2015]. In one of its sections, this document describes “who is doing the work” of developing the Linux kernel. To answer this question, the study ranks the developers by number of changes. It reports, for example, that the top-10 individual developers have contributed with 8.4% of the number of changes. However, the report does not provide information about the specialization of such developers and how frequently they collaborate to solve implementation or evolution tasks.

In summary, code authorship measures can be used to check some important properties and practices in software development, like the ability of an open source system to attract not only new developers, but also new authors; to assess the contributions of paid developers in commercial software (since in this case we should expect all developers to be authors of at least some files); to assess the concentration of knowledge in few team members, which can raise concerns in case they leave the project (which can happen both in open source and in commercial projects); to check whether the criteria used to decompose a system in modules is indeed able to foster the specialization of the work force, as usually expected from software modularization; and to use information on multiple authorship to check practices like collective ownership of the code base, as commonly advocated by agile methodologies [Beck and Andres, 2004].

Linux Kernel Evolution and Conway’s Law. Proposed in 1968, Conway’s Law asserts that “organizations are constrained to produce application designs which are copies of their communication structures” [Conway, 1968]. Although, proposed in the context of formal organizations, it is also worth to investigate whether it applies to more informal organizations, like open source communities. After reasoning on its use in the context of our study, we are inclined to affirm that Conway’s Law does not hold in Linux. Indeed, we collect preliminary evidences that in Linux an *inverse* form of this law better explains the relation between the organization of the Linux development team and the architecture of the system. By inverse, we mean that it is the system’s architecture that shaped Linux’s development team in the last ten years. Linux follows a monolithic architecture, with a *Core* component responsible for its main features [Love, 2010]. The other subsystems provide specialized services, like *Drivers*, *Net*, and *Fs*. This early architectural decision was crucial to define some key characteristics of the Linux’s development team along the years. Our study, for example, shows that Linux has a group of top-authors, who are generalists and therefore work not only in the *Core* but also in other subsystems. By contrast, the remaining authors tend to focus their work in specific subsystems.

2.5 Measuring Code Authorship in a Large Dataset

In this section, we describe an extension of the Linux kernel study, in which we apply the authorship-related metrics proposed in this work to a large dataset. This dataset is composed of 114 open source systems, which are implemented in six different programming languages. These systems come from the truck factor study described in Chapter 3. The original dataset contains 133 systems, but we removed the `torvalds/linux` repository because it was previously analyzed (Section 2.3) and also systems with less than 2 top-level directories (8 repositories) or less than 10 authors (13 repositories).¹³ These last two filters were used to allow, respectively, the computation of the specialization and co-authorship measures.

Figure 2.10 presents the code authorship measures. In the first violin plot (Figure 2.10a) we can observe that the proportion of authors is small in most of the studied systems (the first, second, and third quartiles are 16%, 21%, and 32%, respectively). This behavior was previously observed in the Linux kernel (26%). However, we found systems with a high ratio of authors, which usually are systems with a relevant number of paid developers and some of them are supported by commercial organizations. This

¹³Three systems match the two filters, therefore in total these filters remover 18 repositories.

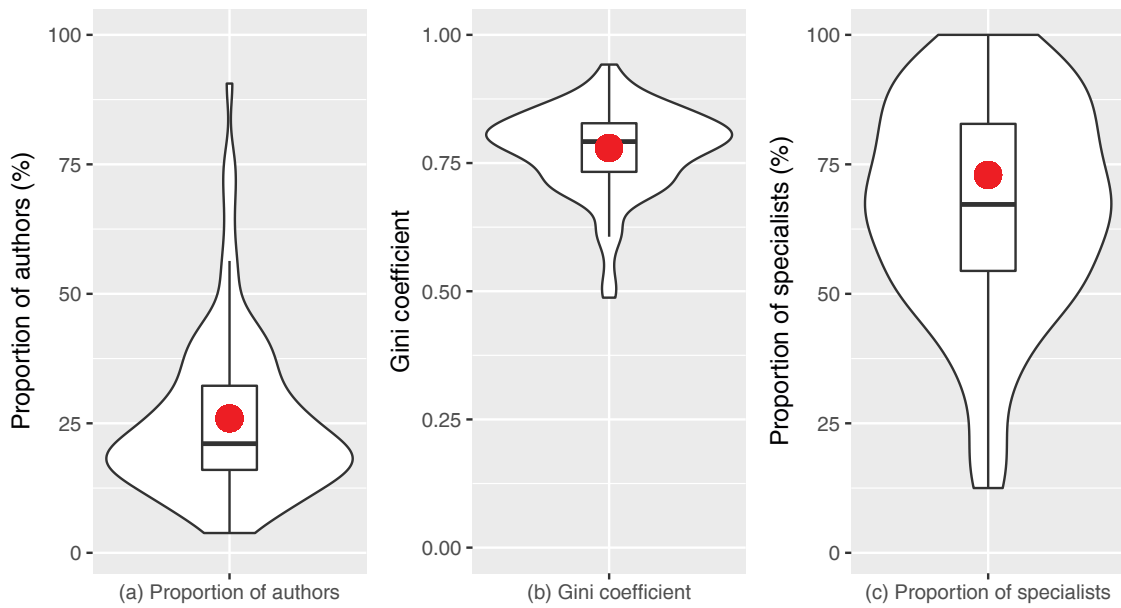


Figure 2.10. Authorship measures in an extended dataset of 114 open source systems. The Linux kernel results are represented by the red dots.

is the case of three out of five outliers, including systems like `v8/v8` (76%), `WordPress/WordPress` (72%), and `JetBrains/intellij-community` (62%). We also detected two language interpreters among the top-6 systems: `ruby/ruby` (68%) and `php/php-src` (56%).

The second violin plot (Figure 2.10b) presents the Gini coefficients. The high coefficients (the first, second, and third quartiles are 0.73, 0.79, and 0.83, respectively) show that the distribution of the number of files per author is highly skewed in most of the systems in the dataset. The systems with a more equal distribution (outliers) are `github/linguist` (0.49), `resque/resque` (0.50), `fzaninotto/Faker` (0.53), and `alexreisner/geocoder` (0.57). Most of these outliers have few authors; however `fzaninotto/Faker`, with 180 authors, is an exception. Its low Gini coefficient, when contrasted with the other systems in the dataset, is a result of its plugin-based software architecture. `fzaninotto/Faker` has a small core and most of the repository’s files are plugins (called *providers*) developed by the community. Again, the Linux kernel’s Gini coefficient (0.78) is close to the median value of the distribution in the extended dataset.

Finally, the third violin plot (Figure 2.10c) presents the proportion of specialists; in this case, we consider as subsystem the top-level directories of the cloned repositories. The high specialization observed in the Linux kernel (73%, by using the directory approach) its also found in most of the systems in our extended dataset (the first, second, and third quartiles are 54%, 67%, and 83%, respectively).

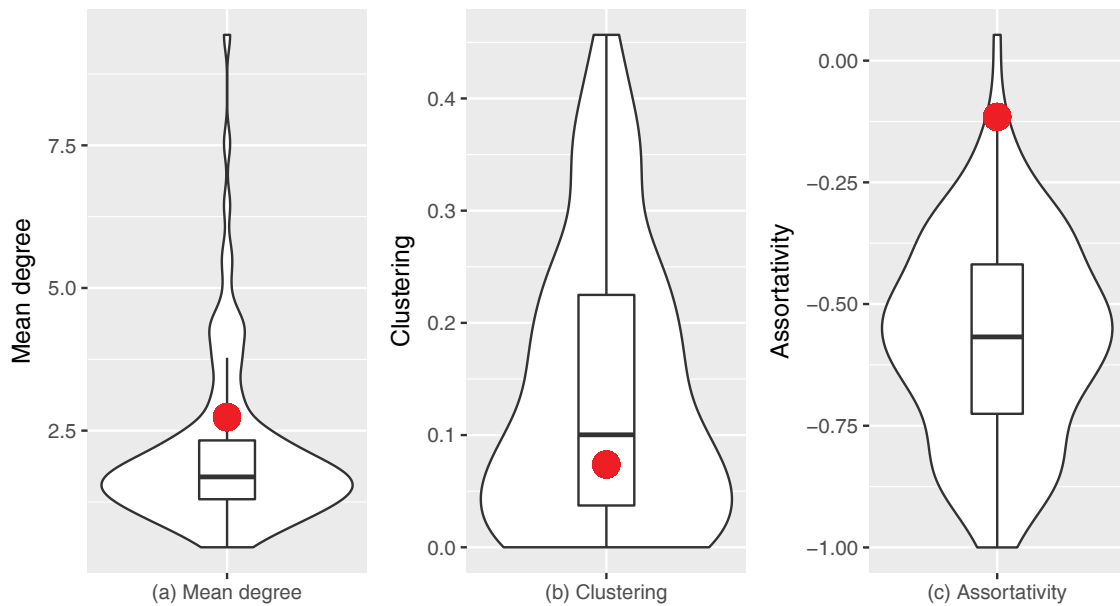


Figure 2.11. Co-authorship measures in an extended dataset of 114 open source systems. The Linux kernel results are represented by the red dots.

Co-authorship measures. We also build the co-authorship network of the 114 systems in the extended dataset. Figure 2.11 presents the co-authorship measures. The first violin plot (Figure 2.11a) shows the mean degree, which is low in most of the systems (the first, second, and third quartiles are 1.29, 1.69, and 2.33, respectively). Similar to what we observed in Figure 2.10a, high mean degrees are more common in repositories supported by commercial organizations—e.g., `JetBrains/intellij-community` (9.44), `v8/v8` (7.53), and `WordPress/WordPress` (4.33)—and in language interpreters—e.g., `php/php-src` (5.33) and `ruby/ruby` (4.43). Although not so high as in these commercial projects, the Linux kernel also presents a high mean degree (2.74), when contrasting with the entire dataset results (mean equal to 2.09).

The clustering coefficient (Figure 2.11b) indicates that most systems have a low probability to form high density co-authorship clusters (the first, second, and third quartiles are 0.04, 0.10, and 0.28, respectively). This trend is also followed by the Linux kernel (0.07).

Finally, the assortativity coefficients are presented in the third violin plot (Figure 2.11c). With the exception of `webscalesql/webscalesql-5.6` (0.05), all the repositories have negative coefficients. In other words, authors with a high number of co-authorship connections tend to collaborate with authors with less connections, suggesting some sort of mentorship. This behavior was previously observed in the Linux kernel (−0.11), but it is more intense in the extended dataset. As example, the clustering coefficients

of `bumptech/glide` and `getsentry/sentry` are -1.00 (the minimal possible value). These two repositories have one main author and all co-authorship connections are between this main author and authors without any additional connections.

Most of the authorship patterns previously observed in the Linux kernel are also common in an extended dataset of 114 popular GitHub open source projects. For example, the repositories in this dataset also have few authors, skewed distributions of the number of files per author, and a high proportion of specialists. Additionally, the co-authorship network measures confirm some initial assumptions about co-authorship patterns, such as: most of the repositories have a low mean degree (but this is higher in projects with commercial support), and it is common some kind of mentorship, where authors with many co-authorship connections are connected with authors with fewer connections.

2.6 Threats to Validity

Construct Validity. With respect to construct validity, our results depend on the accuracy of DOA calculations. Currently, we calculate the degree-of-authorship values using weights from the analysis of other systems [Fritz et al., 2010, 2014]. Although the authors of the absolute DOA claim that their weights are general, we cannot fully eliminate the threat that the choice of weights pose to our results. However, we previously applied them when analyzing different open source systems, obtaining positive feedback from developers [Avelino et al., 2016].

Additionally, our normalized DOA measures may also vary due to possible developer aliases. We mitigate such a threat by handling the most common sources of aliasing—see Section 2.2.3.

Internal Validity. We measure authorship considering only the commit history of the official Linux kernel Git repository. Hence, we do not consider forks that are not merged into the mainstream development. Although these changes might be relevant to some (e.g., studies about integration activities, like rebasing and cherry-picking [German et al., 2015]), they are not relevant when measuring authorship of the official Linux kernel codebase. We also consider that all commits have the same importance when computing authorship. As such, we do not account for the granularity of changes (number of lines of code modified, deleted or inserted) nor their semantics (e.g., bug fixes, new features, refactoring, etc). In open source systems, it is common

that many contributors do not have permission to directly commit their code to the main source code repository. In such scenario, their code, when approved, is integrated by another developer (the committer). Git systems store both information, the author and the committer of the changes. To give credit to the developer who really performed the change, our approach relies on the author information to compute files' authorship.

External Validity. The metrics we use can be applied to any software repository under a version control system. Additionally to Linux kernel we applied the same metrics to a large dataset of open source systems. Although the results confirm that most of the findings identified in the Linux kernel are also common in the extended dataset, we cannot assume that the findings about workload, specialization, and collaboration among file authors are general. In special to not open source systems. Nonetheless, we pave the road for further studies to validate our findings in such development environment.

2.7 Related Work

Code Authorship. McDonald and Ackerman propose the “Line 10 Rule”, one of the first and most used heuristics for expertise recommendation [McDonald and Ackerman, 2000]. The heuristic considers that the last person who changes a file is most likely to be “the” expert. Expertise Browser [Mockus and Herbsleb, 2002a] and Emergent Expertise Locator [Minto and Murphy, 2007] are alternative implementations to the “Line 10 Rule”. A finer-grained approach that assign expertise based on the percentage of lines a developer has last touched are used by Girba et al. [2005] and Rahman and Devanbu [2011]. Similarly, version control systems provide “blame” tools, like `git-blame` [Chacon and Straub, 2014] and `svn-blame` [Pilato et al., 2008]. Essentially, these tools reveal the authors who last modified each line in a file.

Our study relies on the *Degree-of-Authorship* (DOA) metric [Fritz et al., 2010, 2014] to identify developers who perform the most significant contributions to a file. Different from the “Line 10 Rule”, the DOA equation considers the whole version history to compute the degree-of-authorship of a developer with respect to a given code element. DOA is one of the components of the *Degree-of-Knowledge* (DOK) model, which combines authorship and interaction data to identify source code experts. We use only the authorship component because the interaction component requires a plugin to monitor the development environment.

Social Network Analysis (SNA). Research in this area usually extracts information

from source code repositories to build a social network, adopting different strategies to create the links between developers. López-Fernández et al. [2006] apply SNA to study the relationship among developers and how they collaborate in different parts of a project. They use a coarse-grained approach, linking developers that perform commits to the same module. Other studies rely on fine-grained relations, building networks connecting developers that change the same file [Yang, 2014; Meneely et al., 2008; Jermakovics et al., 2011; Bird et al., 2011]. Joblin et al. [2015] propose an even more fine-grained approach. They claim that file-based links result in a dense network, which obscures important network properties, such as community structure. For this reason, they connect developers that change the same function in a source code. Our results, although centered on file-level information, do not produce dense networks, as authorship requires that developers make significant contributions to a file. Consequently, files usually have few authors. Other studies build social networks from mailing lists [Bird et al., 2008; Zhang et al., 2011], issue tracks [Panichella et al., 2014; Hong et al., 2011], or connect developer that work in the same projects, building inter-projects networks [Madey et al., 2002; Xu et al., 2005]. They also investigate the use of collaboration networks to predict failures [Meneely et al., 2008; Pinzger et al., 2008; Bird et al., 2011] and to assess peer review process [Yang, 2014].

Studies on Open Source Development One of the first studies on open source software development was conducted by Mockus et al. [2002]. The authors investigate the Apache and Mozilla ecosystems, including aspects such as developer’s participation, core team size, code ownership, productivity, and defect density. They also compare their results with commercial projects. Vasilescu et al. [2014b] investigate how the workload of the contributors varies across projects in the Gnome community. Recent studies were also conducted using GitHub projects. For example, there is work investigating the impact of GitHub features, such as social coding [Vasilescu, 2014; Vasilescu et al., 2014a], pull requests [Gousios et al., 2014; Tsay et al., 2014] and distributed version control [Barr et al., 2012; Rodriguez-Bustos and Aponte, 2012]. Other studies investigate the influence of programming language on software quality [Ray et al., 2014] and the characteristics of community contributions [Padhye et al., 2014].

In contrast with some of such work, our study does not try to explain the OSS development model or specific characteristics of the GitHub environment. Instead, we propose a set of code authorship based metrics that can be applied to analyze the development and evolution of software projects.

2.8 Conclusion

Seeking to contribute to a better understanding of how authorship-related measures evolve in successful open source communities, we extract and analyze authorship parameters from 1 + 114 systems. Initially, we propose and investigate authorship measures in the Linux kernel. By mining over 11 years of the Linux kernel commit history, we reveal some organizational aspects of the Linux development team, such as authors workload, degree of specialization and co-authorship patterns. Complementary, we apply the same authorship measure to an extended dataset of 114 popular projects and confirm that the authorship patterns observed in the Linux are also followed by other open source systems.

Chapter 3

Estimating Truck Factors

Truck Factor (TF) is a metric proposed by the agile community as a tool to identify concentration of knowledge in software development environments. It states the minimal number of developers that have to be hit by a truck (or quit) before a project is incapacitated. In other words, TF helps to measure how prepared is a project to deal with developer turnover. Despite its clear relevance, few studies explore this metric. Altogether there is no consensus about how to calculate it, and no supporting evidence backing estimates for systems in the wild. To mitigate both issues, we propose a novel (and automated) approach for estimating TF-values, which we execute against a corpus of 133 popular project in GitHub. We later survey developers as a means to assess the reliability of our results. Among others, we find that the majority of our target systems (65%) have $TF \leq 2$. Surveying developers from 67 target systems provides confidence towards our estimates; in 84% of the valid answers we collect, developers agree or partially agree that the TF's authors are the main authors of their systems; in 53% we receive a positive or partially positive answer regarding our estimated truck factors.

3.1 Introduction

A system's truck factor (TF) is defined as “*the number of people on your team that have to be hit by a truck (or quit) before the project is in serious trouble*” [Williams and Kessler, 2003]. Systems with a low truck factor spot strong dependencies towards specific personnel, forming knowledge silos among developer teams. If such knowledgeable personnel abandon the project, the system's lifecycle is seriously compromised, leading to delays in launching new releases, and ultimately to the discontinuation of the project as whole. To prevent such issues, comprehending a system's truck factor is a crucial mechanism.

Currently, the existing literature defines truck factor loosely. For the most part, there is neither a formal definition of the concept nor means to estimate it. The main exception we are aware of stems from the work of Zazworka et al. [2010]. Their definition, however, as well as follow-up works [Ricca and Marchetto, 2010; Ricca et al., 2011], is not backed by empirical evidence from real-world software systems. Stated otherwise, TF-estimates, as calculated by Zazworka’s approach, lack reliability evidence from systems in the wild.

Our work aims to improve the current state of affairs by proposing a novel approach for estimating truck factors, backed up by empirical evidence to support the estimates produced. In particular, we define an automated workflow for TF-estimation, which we apply to a target corpus comprising 133 systems in GitHub. In total, such systems have over 373K files and 41 MLOC; their combined evolution history sums to over 2 million commits. By surveying and analyzing answers from 67 target systems, we provide evidence that in 84% of valid answers, developers agree or partially agree that the TF’s authors are the main authors of their systems; in 53% we receive a positive or partially positive answer regarding our estimated truck factors.

From the work presented in this chapter, we claim the following contributions:

1. A novel approach for estimating a system’s truck factor, as well as a publicly available supporting tool.¹
2. An estimate of the truck factors of 133 GitHub systems. All our data is publicly available for external validation,² comprising the largest dataset of its kind.
3. Empirical evidence of the reliability of our truck factor estimates, as a product of surveying the main contributors of our target systems. From the survey, we also report the practices that developers argue as most useful to overcome a truck factor event.

This chapter is organized as follows. We start by presenting a concrete example of truck factor concerns in the early days of Python development (Section 3.2). Next, in Section 3.3, we present our novel approach for truck factor estimation, detailing all its constituent steps. Next, Section 3.4 discusses our validation methodology, followed by the truck factors of our target systems in Section 3.5. We proceed to present our validation results from a survey with developers (Section 3.6), further discussing results in Section 3.7. We argue about possible threats in Section 3.8. We present the related work in Section 3.9, concluding the study in Section 3.10.

¹<https://github.com/aserg-ufmg/Truck-Factor>

²<http://aserg.labsoft.dcc.ufmg.br/truckfactor>

3.2 Truck Factor: An Example from the Early Days of Python

“What if you saw this posted tomorrow: Guido’s unexpected death has come as a shock to us all. Disgruntled members of the Tcl mob are suspected, but no smoking gun has been found...”—Python’s mailing list discussion, 1994.³

Years before the first discussions about truck factor in eXtreme Programming realms,⁴ this post illustrates the serious threats of knowledge concentration in software development. By posting the fictitious news in Python’s mailing list, the author, an employee at the National Institute of Standards and Technology/USA, wanted to foster the discussion of Python’s fragility resulting from its strong dependence to its creator, Guido van Rossum:

“I just returned from a meeting in which the major objection to using Python was its dependence on Guido. They wanted to know if Python would survive if Guido disappeared. This is an important issue for businesses that may be considering the use of Python in a product.”

Fortunately, Guido is alive. Moreover, Python no longer has a truck factor of one. It has grown to be a large community of developers and the third most popular programming language in use.⁵ However, the message illustrates that Python was, at least by some, considered a risky project. As knowledge was not collective among its team members, but rather concentrated in a single “hero”, in the absence of the latter, discontinuation was a real threat, or at minimum, something that could cause extreme delays. Projects with low truck factor, as Python was in 1994, face high adoption risk, discouraging their use.

To facilitate decision making, it is crucial to quantify the risks of project failure due to personnel lost. This information serves not only business managers assessing early technology adoption risks, but also maintainers and project managers aiming to identify early knowledge silos among development teams. Timely action plans can then be devised as a means to prevent long-term failure. In that direction, reliable estimations of a project’s truck factor is a must.

³<http://legacy.python.org/search/hypermail/python-1994q2/1040.html>

⁴<http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/>

⁵<https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/>, last accessed on June, 2018

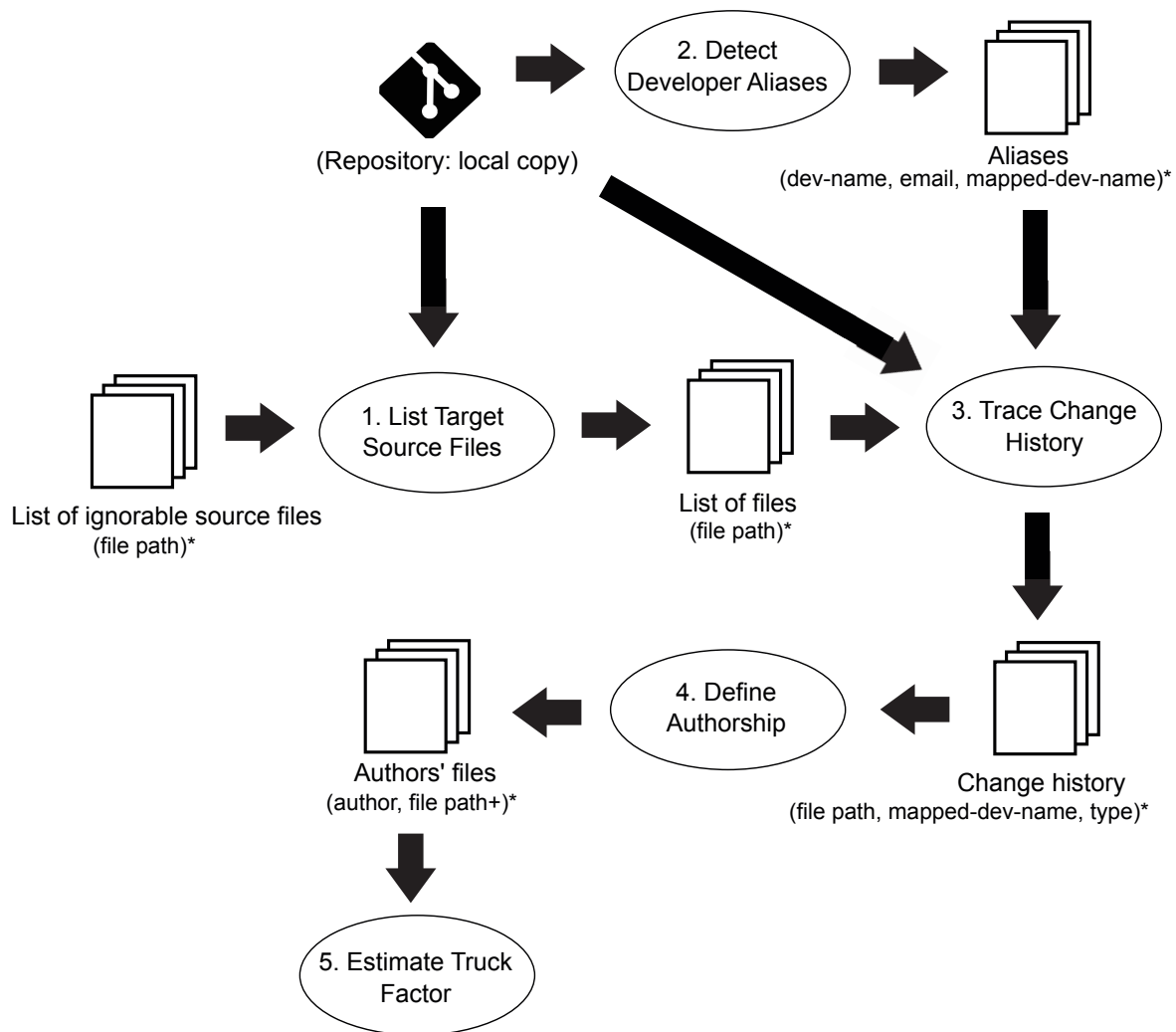


Figure 3.1. Proposed approach for truck factor calculation

3.3 Proposed Approach

The proposed approach calculates the truck factor of a target system by processing its evolution history. We assume the latter to be managed by a version control system, in addition to having access to a local copy of the repository of the target subject.

The process comprises five major steps—see Figure 3.1. Step 1 checkouts the latest point in the commit history, listing all the source files therein. Step 2 handles possible aliases among developers, *i.e.*, cases where a single developer has multiple Git users. Step 3 traces the history of each source code file. From such traces, step 4 identify the authors in the system, as well as their authored files. With the list of authors and their authored files, step 5 estimates the truck factor of the entire system.

We execute the given process to automatically estimate the truck factor of

projects whose evolution is managed by Git. In the following, we detail each step.

3.3.1 Main Steps

Step 1: List Target Source Files. To obtain the list of target files, first, the process switches to the master branch of the target repository, checking out its latest commit. Then, it enumerates the path of all source files of the given snapshot, excluding all other file types (*e.g.*, files representing documentation, images, examples, etc), as well as the files listed in the ignorable source file list, given as input. It also discards source files associated with third-party libraries (*i.e.*, files that are not developed in the system under analysis). Our decision is conservative. An existing survey from JavaOne’14⁶ reports that nearly two-thirds of polled senior IT professionals have Java applications with half of their code coming from third-party sources. Thus, if developers store third-party code in the system’s main Git repository (*e.g.*, as backup, to facilitate build, etc), and third-party code is as large as the poll suggests, truck factor estimates are likely to be significantly affected.

Excluding third-party code requires being able to identify it in the first place. As such, our approach employs Linguist,⁷ an opensource tool from GitHub. Linguist is actively developed, and it is constantly being updated by the GitHub community to include new pattern matching rules to identify third-party file names. Linguist’s original goal is to detect the programming language of GitHub projects—as in our case, this is sensitive to external code.

Step 2: Detect Developer Aliases. Each user in Git is a pair (*dev-name*, *email*)—*e.g.*, (“Bob Rob”, “bob.rob@example.com”). It happens, however, that a single developer may be associated with many Git user accounts, leading to *developer aliases*. To handle aliases our approach applies the same strategy described in Section 2.2.3. As a result of alias detection, step 2 outputs a mapping from Git users to a single developer name (*mapped-dev-name*).

Step 3: Trace Change History. This step traces the evolution history of each target file, taking as input the results of the previous two steps. To perform the tracing, our approach collects the system’s commits using the `git log -find-renames` command. This command returns all commits of a repository and identifies possible file renames. Then, it processes each commit, extracting three pieces of information: (i) the path of the file we are collecting the trace; (ii) the *mapped-dev-name* of the developer performing the change; and (iii) the type of the change—file addition, file modification, or file

⁶<http://tinyurl.com/javaone14-survey>

⁷<https://github.com/github/linguist>

rename.

Step 4: Define Authorship. Given the change traces of each file in the target snapshot of the project at hand, this step defines the author list of each file. Different alternatives could be used as a means for determining authorship—*e.g.*, [Anvik et al., 2006; Minto and Murphy, 2007; Schuler and Zimmermann, 2008; Hattori and Lanza, 2009; Fritz et al., 2010, 2014]. Among those, we chose the *degree-of-authorship* (DOA) metric [Fritz et al., 2010, 2014], which is normalized after calculation. Details about how we adapt the DOA metric to identify the authors of a file were previously described in Section 2.2.1. As a result, this step outputs a list of associations from authors (*mapped-dev-names*) to their related authored files.

Step 5: Estimate Truck Factor. Taking a list A of authors (*mapped-devs*) and their associated authored files (one or more file paths), this step estimates the system’s truck factor. Our estimation relies on a coverage assumption: a system will face serious delays or will be likely discontinued if its current set of authors covers (i.e., authored) less than 50% of the current set of files in the system. Following such assumption, our truck factor estimation algorithm implements a greedy heuristic—see Algorithm 1. Starting with a truck factor of zero, the algorithm iterates over the authors’ file list A (lines 4–11), verifying at each iteration whether the current authors’ coverage is below 0.5 (line 6). If so, the algorithm stops the iteration—maintenance is likely to be hampered; otherwise, it removes the top author from A (line 9), increasing truck factor by one (line 10). The top author in a given iteration is the *mapped-dev* authoring the highest number of files in A .⁸ Whenever A shrinks, another iteration follows, provided A is not empty. This process continues until A becomes empty or coverage is less than 0.5.

3.4 Validation Methodology

To validate our approach, we select 133 systems from GitHub. For each target system, we estimate its truck factor. This section details our corpus selection and how we setup our approach for estimating truck factors for our chosen subjects. We also discuss how we survey developers as a means to validate our estimates and get further insights.

⁸This is obtained by finding the entry $e_i = (a_i, \text{filepath-list}_i) \in A$ s.t. $\nexists e_j = (a_j, \text{filepath-list}_j) \in A \wedge e_j \neq e_i \wedge |\text{filepath-list}_j| > |\text{filepath-list}_i|$. If there exist more than one top author, we just take the first one we find.

Algorithm 1: TRUCK FACTOR ALGORITHM.

```

Input: List of authors' files  $A$ 
Output: System truck factor
1 begin
2    $F \leftarrow \text{getSystemFiles}(A)$ ;
3    $tf \leftarrow 0$ ;
4   while  $A \neq \emptyset$  do
5      $\text{coverage} \leftarrow \text{getCoverage}(F, A)$ ;
6     if  $\text{coverage} < 0.5$  then
7       break;
8     end
9      $A \leftarrow \text{removeTopAuthor}(A)$ ;
10     $tf \leftarrow tf + 1$ ;
11  end
12  return  $tf$ ;
13 end

```

3.4.1 Selection of Target Subjects

To select a target set of subjects, we follow a procedure similar to other studies investigating GitHub [Yamashita et al., 2015; Gousios et al., 2014; Kalliamvakou et al., 2014; Ray et al., 2014]. First, we query the programming languages with the largest number of repositories in GitHub. We find six main languages (L): JavaScript, Python, Ruby, C/C++, Java, and PHP. We then select the 100-top most popular repositories within each target language (as collected from GitHub on February 25th, 2015). Popularity, in this case, is given by the number of times a repository has been starred by GitHub users. Considering only the most popular projects in a given language (S_ℓ), we remove the systems in the first quartile (Q_1) of the distribution of three metrics, namely number of developers (n_d), number of commits (n_c), and number of files (n_f). After filtering out subjects in Q_1 , we compute the intersection of the remaining sets. From the previous steps, we get an initial set of prospective subjects T^0 . Formally,

$$T^0 = \bigcup_{\ell \in L} T_{n_d}^0(\ell) \cap T_{n_c}^0(\ell) \cap T_{n_f}^0(\ell)$$

where

$$\begin{aligned} T_{n_d}^0(\ell) &= S_\ell - Q_1(n_d(S_\ell)), & T_{n_c}^0(\ell) &= S_\ell - Q_1(n_c(S_\ell)), \\ T_{n_f}^0(\ell) &= S_\ell - Q_1(n_f(S_\ell)) \end{aligned}$$

From T^0 , we determine a new subset T^1 including only the systems whose repositories stem from a proper migration to GitHub. Specifically, we remove systems with more than 50% of their files added in less than 20 commits—less than 10% of the minimal number of commits we initially considered. This evidences that a large portion

Table 3.1. Target repositories

Language	Repos	Devs	Commits	Files	LOC
JavaScript	22	5,740	108,080	24,688	3,661,722
Python	22	8,627	276,174	35,315	2,237,930
Ruby	33	19,960	307,603	33,556	2,612,503
C/C++	18	21,039	847,867	107,464	19,915,316
Java	21	4,499	418,003	140,871	10,672,918
PHP	17	3,329	125,626	31,221	2,215,972
Total	133	63,194	2,083,353	373,115	41,316,361

of a system was developed using another version control platform and the migration to GitHub could not preserve the original version history. From the resulting set of prospective subjects ($|T^1| = 135$), we manually inspect the documentation in each repository to identify and eliminate duplicate subjects. Our inspection shows `raspberrypi/linux` and `django/django-old` as duplicate cases. The first, despite not being a fork, is very similar to `torvalds/linux`; in fact, it is a clone of the Linux kernel, with extensions supporting RaspberryPi-based boards. The second is an old version of a repository already in T^1 .

After excluding `raspberrypi/linux` and `django/django-old`, we are left with 133 subjects (T^2), which represent the most important systems per language in GitHub, implemented by teams with a considerable number of active developers and with a considerable number of files. Table 3.1 summarizes the characteristics of the repositories of our chosen subjects. Ruby is the language with more systems, 33 in total. The programming language with less systems is PHP, with 17 projects. Accounting all our chosen subjects, their latest snapshots accumulate over 373K files and 41 MLOC; their combined evolution history sums to over 2 million commits. Our targets also have a large community of contributors, accumulating to over 60K developers. The violin plots in Figure 3.2 depict each distribution.

3.4.2 Setting up Inputs

Our approach requires as input a listing of ignorable source files of a system, in addition to a tuple of the DOA thresholds (k and m). Next, we detail how we set both inputs.

List of Ignorable Source Files. To create the list of ignorable files, we manually inspect the first two top-level directories in each target repository, seeking to find third-party libraries undetected by Linguist. Also, as Linguist is architecture and system agnostic, we look for plugin-related code in systems with a plugin-based architecture. As with

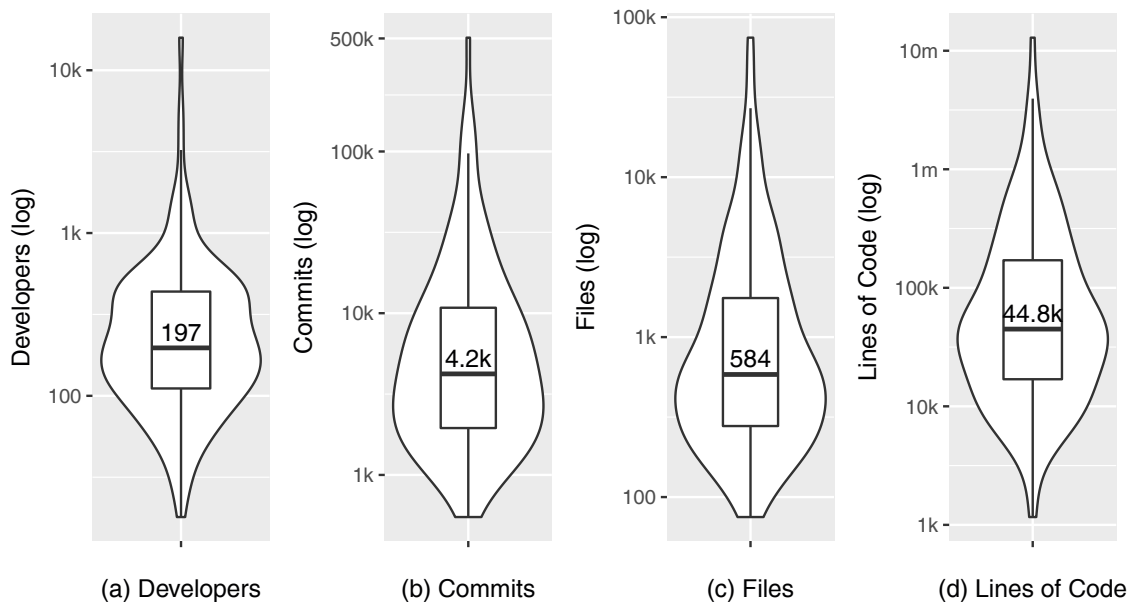


Figure 3.2. Target subjects

third-party code, plugins may highly influence a system’s truck factor. For instance, in the Linux kernel, driver plugins are the most common feature type [Passos et al., 2015]; since driver features generally denote optional features targeting end-user selection, the kernel itself is independent from them. In the case of `torvalds/linux`, we exclude all driver-related code, which is, for the most part, inside the `driver` folder of the Linux kernel source code tree.⁹

In addition to the Linux kernel repository, two other systems have a large amount of plugin-related code: `Homebrew/homebrew` and `caskroom/homebrew-cask`. Homebrew is a package manager in Mac OS for handling the installation of different software systems. Its implementation allows contributors to push new formulas (automated installation recipes) to the system’s remote repository, leading to thousands of formulas. As an extensible software system, Homebrew has one of the largest base of developers on GitHub (more than 5K developers, as of July 14th, 2015). Considering all its formulas, Homebrew’s TF, as computed by our heuristic, is 250. After excluding the files in folder `Library/Formula`, however, HomeBrew’s truck factor reduces to 2. This clearly evidences the sensitivity of TF-values in the face of external code. As for `caskroom/homebrew-cask`, we ignore its `Casks` directory.

In total, our list of ignorable files excludes 10,450 entries.

⁹Specifically, we identify all driver-related code by executing a specialized script from G. Kroah-Hartman, one of the main developers of the Linux kernel. Available at <https://github.com/gregkh/kernel-history>.

Setting DOA Thresholds. To find suitable thresholds to the Equation 2.3, we manually inspect a random sample of 120 files stemming from the six most popular systems in our target corpus (T^2), one for each target language we account for. This results in files from `mbostock/d3` (JavaScript), `django/django` (Python), `rails/rails` (Ruby), `torvalds/linux` (C/C++), `elasticsearch/elasticsearch` (Java), and `composer/composer` (PHP). We then compute the normalized DOA values (DOA_N) for each developer contributing at least one commit changing a file in our sample. Initially, we note that DOA_N values below 0.50 lead to doubtful authorships. We measure doubtfulness by contrasting our authorship results with ranks we extract from `git-blame` reports. The latter contains the last developer who modified each line in a file [Chacon and Straub, 2014]; by ranking developers according to the number of their modified lines, authors are likely to be those with higher ranks. Fixing 3.293 (which corresponds to the constant term in DOA’s linear equation) as minimal absolute DOA (DOA_A) and resetting the threshold for DOA_N to 0.75 better aligns results. Specifically, 64% of the authors selected using those thresholds are classified as the top-1 ranked developer from `git-blame`; in 91% of the cases, they are among the top-3 in the ranking list of `git-blame`, whereas 7% lie between the 4th and 8th positions. In only three cases (2%), the authors do not pair with any developer from `git-blame` rankings.

3.4.3 Survey Design and Application

After collecting the truck factors of our chosen targets, we set to elaborate survey questions aiming to confirm the reliability of our results, as well as an instrument to get further insights. Following best practices in survey design [Shull et al., 2007], we assure clarity, consistency, and suitability of our questions by running a feedback loop between the survey author and two other members of our research group. We also perform a pilot study to identify early problems, such as whether our language correctly captures the intent of our questions. From the pilot study, we note few, but important communication issues, which we fix accordingly.

Survey Questions. After our pilot study, we phrase our questions as follows.

Question 1. Do developers agree that top-ranked authors are the main developers of their projects?

This question seeks to assess the accuracy of our top authorship results. The top-ranked authors of a system are those we remove during the iteration step of our greedy-heuristic (recall Algorithm 1), *i.e.*, those responding for a system’s truck factor. Note that we use the term *main developers*, not authors. Our pilot study shows that developers tend

to consider the creator of a file as its main author.

Question 2. Do developers agree that their project will be in trouble if they lose the developers responsible for its truck factor?

This question aims to validate our TF estimates. If we receive a positive feedback in this question, we can conclude that code authorship is an effective proxy.

Question 3. What are the development practices and characteristics that can attenuate the loss of the developers responsible for a system’s truck factor?

Our intention here is to reveal the instruments developers see as most effective to circumvent the loss of important developers—*e.g.*, by devising better documentation, codification rules, modular design, etc.

Survey Application. Before contacting developers and applying our survey, we aim at calling their attention by promoting our work in popular programming forums (*e.g.*, Hacker News) and publishing a preprint at PeerJ (<https://peerj.com/preprints/1233>).

After promotion, we apply the survey by opening GitHub issues in all target projects allowing such a feature (114 out of 133). The choice for issues is twofold: (i) issues foster public discussions among project developers; (ii) issues document all discussions, making them available for later reading. Potential readers include new developers, end-users interested on the target systems, researchers, etc.

Our posting period ranges from July 31th to August 11th, 2015. In the following weeks, we set to collect answers, respected the deadline of August 25th, 2015. In total, we collect answers from developers of 67 systems. In 37 of those, there is a single answer from a single developer. However, often, the issues include discussions among different project members. For example, in `saltstack/salt`, we have comments from six developers. In total, we accumulate 170 discussion messages from 106 respondents; 96 messages stem (57%) from the top-10 contributors of the 67 participating projects. Figure 3.3 characterizes all participants according to their level of project contribution. We get the list of top contributors by consulting the project’s statistics as provided by GitHub. To exclude unreliable answers, we discard issues that do not have a single answer from a top-10 project contributor—five issues in total. Thus, we are left with 62 participating systems, as we have one issue per system. Among the issues that we do not exclude from analysis, we find 96 different respondents. Some messages are quite detailed. For instance, a message in an issue in `elastic/elasticsearch` contains 1,670 words.

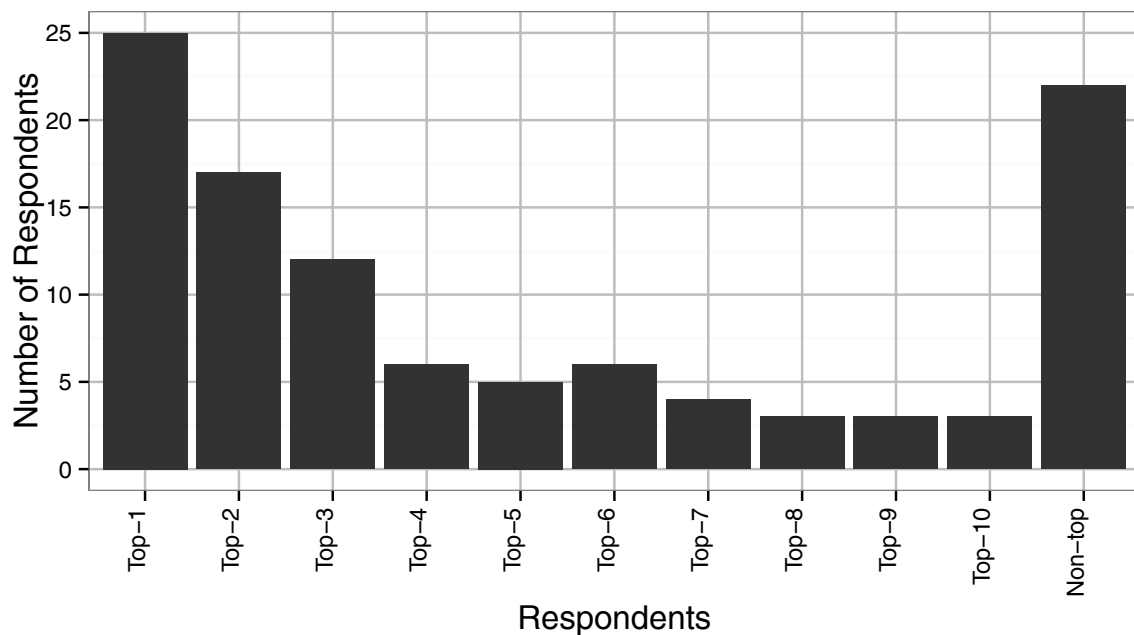


Figure 3.3. Respondents profile

In fact, according to the respondent, it triggered interesting internal discussions.

Survey Analysis. To compile the survey results, we analyze the discussions of our opened issues. For the first two questions, we classify answers according to four levels: agree, partially agree, disagree, or unclear. The fourth level refers to cases where we cannot derive a clear position from an answer. Two authors of this study independently classified all answers, later crosschecking their results.

As for our third question, we categorize answers to identify common practices and characteristics.

3.5 Truck Factor Estimates

In this section, we describe the outputs of applying the proposed approach on the 133 projects of the selected dataset.

3.5.1 Preceding Output

Target List of Source Files (Step 1). Using our input list of ignorable files (see Section 3.4), as well as the automated exclusion by Linguist, we estimate the authorship of 243,660 files (33 MLOC)—34% less files than the original set in our subjects. The most frequent kind of files we remove concern JavaScript (5,125), PHP (3,099), and C/C++ (2,049) source files. Decreasing the number of target files decreases the target

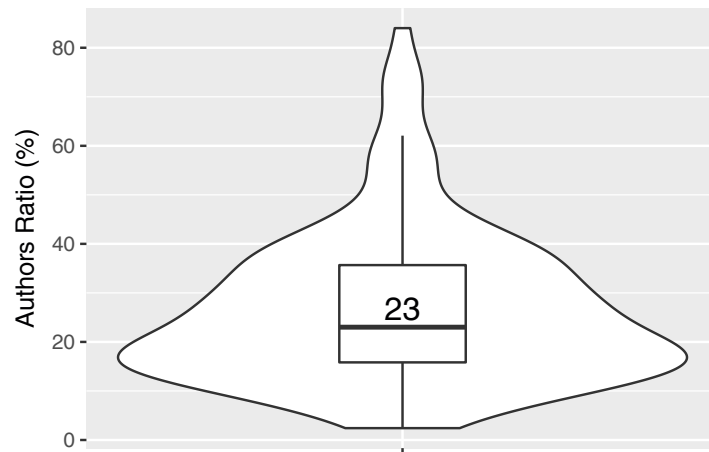


Figure 3.4. Proportion of developers ranked as authors

number of developers (63,193) and commits (1,262,130), a reduction of 28% and 39% w.r.t the original state of our target repositories.

Authorship List (Step 4). By applying the normalized DOA to define the list of authors in each target system, as well as their authoring files, step 4 reveals the proportion of developers ranked as authors—see Figure 3.4. For most systems, such proportion is relatively small; the first, second, and third quartiles are 16%, 23%, and 36%, respectively. Interestingly, systems with a high proportion of authors usually have support of private organizations. Examples include four of the top-10 systems with the highest author ratio among developers, such as `v8/v8` (75%), `JetBrains/intellij-community` (73%), `WordPress/WordPress` (67%), and `Facebook/osquery` (62%). We also detect two language interpreters among the top-10 systems: `ruby/ruby` (72%) and `php/php-src` (59%). At the other extreme, there are systems with a very low author ratio—*e.g.*, `sstephenson/sprockets` (3%) and `jashkenas/backbone` (2%). Backbone is an interesting example, with only six authors amongst its 248 developers. These six authors monopolize 67% of commits. A similar situation occurs with `sprockets` (a Ruby library for compiling and serving web assets): although 61 developers associate to commits in the evolution history, 95% of commits come from two authors only; moreover, 27 developers respond for a single commit modifying a single line of code.

3.5.2 Results

Figure 3.5 presents the distribution of the truck factor amongst our subjects. The first, second, and third quartiles are 1, 2, and 4, respectively. Most systems have a small

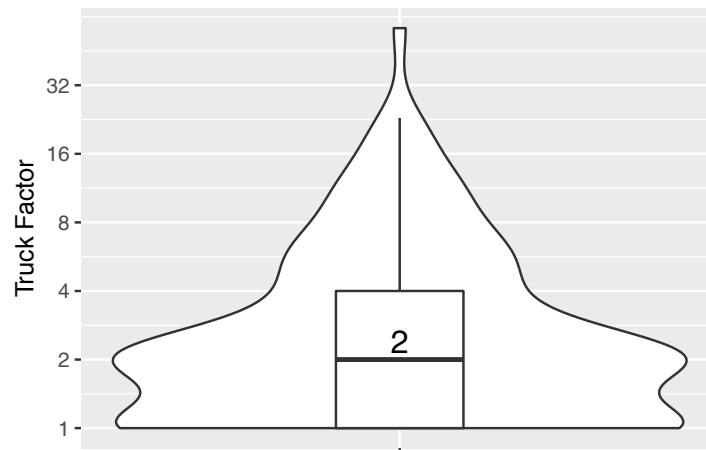


Figure 3.5. Systems Truck Factor

Table 3.2. Systems with highest truck factors

System	TF
torvalds/linux	57
fzaninotto/Faker	23
android/platform_frameworks_base	19
moment/moment	19
php/php-src	18
odoo/odoo	14
fog/fog	12
git/git	12
webscalesql/webscalesql-5.6	11
v8/v8	11
Seldaek/monolog	11
saltstack/salt	11
JetBrains/intellij-community	9
rails/rails	9
puppetlabs/puppet	9

truck factor: 45 systems (34%) have $TF = 1$ (e.g., mbostock/d3 and less/less.js); in 42 systems (31%), $TF = 2$, including well-known systems such as clojure/clojure, cucumber/cucumber, ashkenas/backbone and elasticsearch/elasticsearch. Systems with high TF-values, however, do exist. Table 3.2 presents the top-15 systems with the highest truck factors. Among those, torvalds/linux has $TF = 57$, followed by fzaninotto/Faker ($TF = 23$) and android/platform_frameworks_base ($TF = 19$). Other well-known systems include php/php-src ($TF = 18$), git/git ($TF = 12$), v8/v8 ($TF = 11$), and rails/rails ($TF = 9$).

Table 3.3. Answers for Survey Question 1

Agree	Partially	Disagree	Unclear
31 (50%)	18 (29%)	9 (15%)	4 (6%)

3.6 TF Validation: Surveying Developers

We present our survey results from our filtered set of issues and their underlying messages—we only account for issues having at least one message from a top-10 project contributor. In total, the answers we analyze stem from 106 respondents, of which 84 are top-10 contributors. The final number of participating systems is 62.

Question 1. Do developers agree that the top-ranked authors are the main developers of their projects?

Table 3.3 summarizes the answers for our first question. Respondents of 31 systems (50%) fully *agree* with our list of main developers. Example agreements:

“Yes, that’s me.”—developer from (bjorn/tiled).

“I think that it is a reasonable statement to make. They have contributed by far the most and paved the way for the rest of us.”—developer from (composer/composer).

Developers of 18 systems (29%) *partially agree* with our list of top-ranked authors. The main disagreement stems from the historical balance between older and newer developers:

“Yes and no, historically yes, currently no, a team has been picking up the activity, your analysis seems to be biased on capital (existing files) rather than activity (current commits).”—developer from (kivy/kivy).

“I would have added @DayS and @WonderCsabo as main developers.”—developer from (excilys/androidannotations).

The latter answer illustrates a situation where we report two top-authors in a target project; the respondent, although agreeing with our suggestion, recommends adding two other developers. The latter two have many recent commits; in contrast, one of the top authors we recommend is no longer active, strengthening the developer’s argument. The two top-authors from our degree-of-authorship measures cover 41% and 26% of files, respectively. The two suggested by our respondent account for 9% and 17% (see Figure 3.6). However, we do note a gradual decrease in the number of authored files by the top developer we suggest, while an increasing trend for one of the two that our

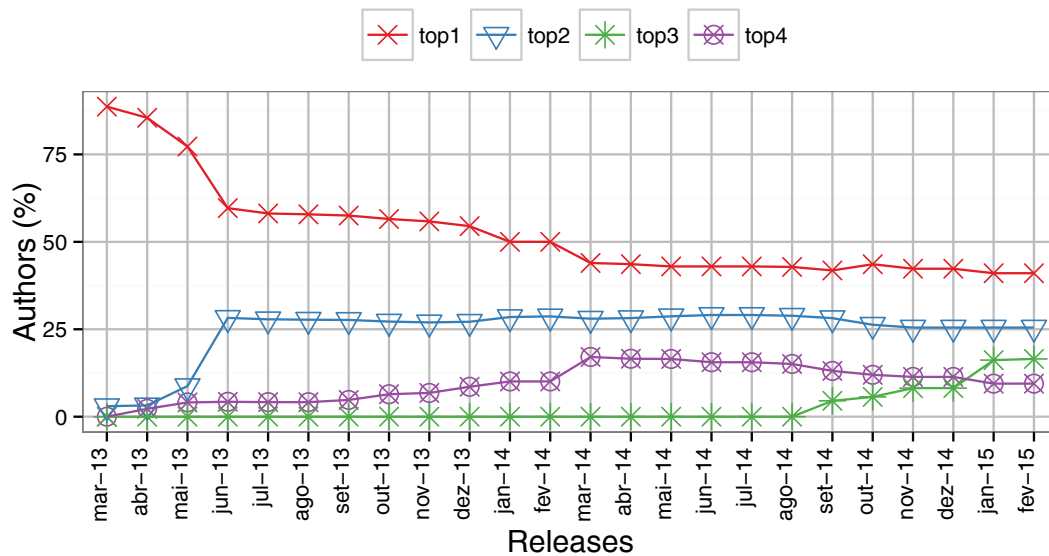


Figure 3.6. Percentage of files per author in excilys/androidannotations (top-4 authors)

respondent recommends.

Developers of nine systems (15%) *disagree* with our list. Six developers indicate that other contributors are now responsible for their projects. Example disagreements include:

“No. TJ has been away from Jade for quite some time now. @ForbesLindesay is considered the main maintainer/developer of Jade.”—developer from (jadejs/jade).

“No, Burns hasn’t been contributing for a while now. I’ve taken over what he was doing.”—developer from (backup/backup).

Other disagreements are due to auto-generated code. Finally, a single developer has a negative attitude towards the question, providing us with no insights.

Question 2. Do developers agree that their projects will be in trouble if they loose the truck factor authors?

Table 3.4 summarizes results concerning this question. Developers of 24 systems (39%) *agree* with our truck factor results. Most positive answers are concise, usually a straight “yes” (14 answers). We consider as agreement answers that acknowledge a serious impact to the project if the given developers are to be absent, such as in:

“If both of us left, the project would be kind of unmaintained.”—developer from (SFTtech/openage).

“Initially, yes. However, given the size of the Grunt community, I believe a new

Table 3.4. Answers for Survey Question 2

Agree	Partially	Disagree	Unclear
24 (39%)	6 (10%)	27 (43%)	5 (8%)

maintainer could be found.—developer from (gruntjs/grunt).

“If Wladimir or Pieter left, it would be a serious loss, but not fatal I think.”—developer from (bitcoin/bitcoin).

Among the positive answers, we find a system that in fact “lost” its single truck factor author—pockethub/PocketHub. The project implements a GitHub Android client, originally released as part of the GitHub platform. A GitHub employee is identified as the system’s single author, accounting for 78% of all source files. However, as stated in the repository home page, GitHub no longer maintains the app. The repository is almost inactive, receiving very few commits per month. The last release dates from February 2014 (still as a GitHub project). One developer reports that low community involvement is the reason for the project’s trouble, as there are only three people working on the project and this is not their full time job.

Six answers are *partial agreements*. Examples:

“Somewhat agree. A loss in one area would mean a temporary dip in maintenance of that area until someone else stepped in.”—developer from (saltstack/salt).

“Not necessarily, there’s a long list of both small and significant contributors that had to understand a large piece of the code base to implement a feature or fix.”—developer from (justinfrench/formtastic).

Developers of 27 systems *disagree* with our TF-values. Six developers (22%) have negative answers to our question, but do not provide further details; 21 developers (78%) justify their answer stating that others could take over the project:














“Backup shouldn’t be in trouble...It’s an open source project, anyone can start contributing if they want to.”—developer from (backup/backup).

We find two systems surviving the “loss” of the truck factor authors in our list:

“Coda was the author of the majority of the code. He left the project around a year ago. Some issues were going a long time without resolution, at which point I offered to maintain the project.”—developer from (dropwizard/metrics).

“Your questions are timely, since Roland [the main author] has already left the project ... and we are not in trouble.”—developer from (caskroom/homebrew-cask).

Table 3.5. Practices to attenuate the truck factor

Practice	Answers
Documentation	36 
Active community	15 
Automatic tests	10 
Code legibility	10 
Code comments	7 
Founding/Paid developers	5 
Popularity	5 
Architecture and design	4 
Shared repository permissions	4 
Other implementations	2 
Knowledge sharing practices	2 
Open source license	2 
Miscellaneous	9 

In the case of `dropwizard/metrics`, it has been partially affected by the loss of its single truck factor author, as another developer was able to take over the project. As for `caskroom/homebrew-cask`, the respondent highlights two factors helping their transition after losing their single truck factor author: (a) comprehensive documentation; (b) developers ready to transmit the rules and requirements to newcomers.

Question 3. What are the development practices that can attenuate the loss of top-ranked authors?

Table 3.5 summarizes answers. Documentation is the practice with the largest number of mentions across answers (36 answers), followed by the existence of an active community (15 answers), automatic tests (10 answers), and code legibility (10 answers). We group practices with a single answer under the *miscellaneous* category—*e.g.*, implementation in a specific programming language, periodic team chats, code reviews, support to classical algorithms, etc. In addition, we received seven “yes/no” answers, which are nonsensical given the nature of the question (not shown).

Although not development practices, having active communities and paid developers appear frequently among the answers we analyze. Both reasons appear as justifications for not concerning with top-authors lost:

“I’d say that the vibrant community is the reason for it.”—developer from (`rails/rails`).

“We have a handful of other maintainers and a large body of contributors who are interested in Homebrew’s future.”—developer from (`Homebrew/homebrew`).

*“The people you listed are paid to work on the project, along with a number of others. So if the four of us took off, the project would hire some more people”—*developer from (ipython/ipython).

3.7 Discussion

In this section, we discuss the lessons we learn in our study. We also lay out directions for future research on truck factor measurements and applications.

3.7.1 DOA Results

The results produced by the DOA model seem accurate when applied to a large collection of systems. For the first survey question, the developers of 49 systems (84% of the valid answers) agree or partially agree with our results. Despite that, some developers report that the model gives high emphasis on first authorship (FA) events. In the same question of our survey, six developers disagree with our results exactly due to this resilience of the DOA model in transferring authorship from the first author to another one. This applies in systems where a single developer creates the bulk of the code, but later switches role (*e.g.*, project leader or mentor), becoming less active in development activities. In fact, some developers suggest that DOA computation should consider only the most recent development history, *e.g.*, commits performed in the last year. One developer from clojure/clojure explicitly declares that *“if the code is old enough, even the original author will have to approach it with essentially fresh eyes.”*

3.7.2 Challenges on Computing Truck Factors

We receive answers for 67 (out of 114) systems. This high response ratio (59%) is certainly a consequence of the importance that developers give to the truck factor concept. By analyzing the answers, we see that developers generally recognize the impact that the truck factor may have in the public reputation of their systems. However, it is worth noting that estimating this concept automatically has many challenges. A few developers refused to answer our question, stating for example that *“it is an existential, speculative question that I will not attempt to answer”* (developer from mbostock/d3). A second developer states that *“the truck factor is mostly concerned with institutional memory getting lost. No automatic system can account for this lost, unless all project communication is public.”* (developer from libgdx/libgdx). Our survey also reveals that

developers usually consider documentation as the best practice to overcome a truck factor episode.

Despite the challenges in computing truck factors automatically, our code-authorship-coverage heuristic presents compelling results. We receive positive or partially positive answers for 30 systems (53% of the valid answers). Even when developers do not agree with our estimation, it is not completely safe to discard a possible damage to the system. For example, six developers state that truck factor is not a concern in open source systems, since it is always possible to recruit new core developers from their large base of contributors. In fact, we observe a successful transition of core developers in at least two systems. In contrast, we cannot discard the risks inherent to such transitions, specially when they should take place due to a sudden and unplanned truck-factor-like episode.

Developers also pointed two concrete problems in our heuristic for computing truck factors. First, it considers all files in a system as equally important in terms of the features they implement. However, not all requirements and features are equally critical to a system survivability. We address this problem by discarding some files from our analysis, in the cases they lead to highly skewed results (*e.g.*, recipes from Homebrew/homebrew). However, in other systems this partition between core and non-core files is less clear (at least, to non-experts). Second, the heuristic does not account the last time a file is changed. In the survey, some developers claim that losing the author of a very stable file is not a concern (since they probably will not depend again on this author to maintain the file). When such files are common in a system, the heuristic can be adapted to just consider recently changed files.

3.8 Threats to Validity

Construct Validity. We compute the degree-of-authorship using weights derived for other systems [Fritz et al., 2010, 2014]. Therefore, we cannot guarantee these weights as the most accurate ones for assessing authorship on GitHub projects. However, the authors of the DOA formula show that the proposed weights are robust enough to be used with other systems, without computing a new regression. Still, we mitigate this threat by initially inspecting the DOA results for 162 pairs of authors and files. Contrasting results with those from `git-blame` suggest DOA-values to be reliable.

The presence of non-source code files, third party libraries, and developers aliases can also impact our results. To address these threats, our tool performs file cleaning and alias handling steps before calculating truck factor estimates.

Internal Validity. Our approach computes the authors of a file by considering all the changes performed in the target file. Therefore, our approach is sensitive to loss of part of the development history as result of a erroneous migration to GitHub. We mitigate this threat using a heuristic to detect systems with clear evidence that most of its development history was performed using another version control platform and that this history could not be correctly migrated to GitHub. Moreover, the full development history of a file can be lost in case of renaming operations, copy or file split (*e.g.*, as result of a refactoring operation like *extract class* [Fowler, 1999]). We address the former problem using Git facilities (*e.g.*, `git log -find-renames`). However, we acknowledge the need for further empirical investigation to assess the true impact of the other cases.

External Validity. We carefully select a large number of real-world systems coming from six programming languages to validate our approach. Despite these observations, our findings—as usual in empirical software engineering—cannot be directly generalized to other systems, mainly closed-source ones. Many others aspects of the development environment, like contribution policies, automatic refactoring, and development process may impact the truck factor results and it is not the goal of this study to address all of them.

Finally, to assess the impact of the aforementioned threats in our results, we conducted a survey with developers of the systems under analysis, as reported in Section 3.6.

3.9 Related Work

Although widely discussed among *eXtreme Programming* (XP) practitioners, there are few studies providing and validating truck factor measures for a large number of systems. Zazworka et al. [2010] are probably the first to propose a formal definition for TF, specifically to assess a project’s conformance to XP practices. For the purpose of simplicity, their definition assumes that all developers who edit a file have knowledge about it. Furthermore, they only compute the TF for five small projects written by students. Ricca and Marchetto [2010]; Ricca et al. [2011] use Zazworka’s definition to compute truck factors for opensource projects. In their first work, they propose the use of the TF algorithm as a strategy to identify “heroes” in software development environments. In their second work, the authors point for scalability limitations in Zazworka’s algorithm, which only scales to small projects (≤ 30 developers). In our study, 122 out of 133 systems have more than 30 developers (maximum is `torvalds/linux`, with thousands of developers among non-driver files). Hannebauer and Gruhn [2014] further explore

the scalability problems of Zazworka’s definition, showing that its implementation is NP-hard. Cosentino et al. [2015] propose a tool to calculate TF for Git-based repositories. They use a hierarchical strategy, aggregating file-level authorship results to modules and, in a second step, aggregating module-level results into systems. They evaluate their tool with four systems developed by members of their research group.

Overall, our study differs from the previous ones in three main points: we use the DOA model to identify the main authors of a file; we evaluate our approach in a large dataset composed of real-world software from six programming languages; we validate our results with expert developers.

3.10 Conclusion

This study proposes and evaluates a heuristic-based approach to estimate a system’s truck factor, a concept to assess knowledge concentration among team members. We show that 87 systems (65%) have $TF \leq 2$. We validate our results with the developers of 67 systems. In 84% of the valid answers, respondents agree or partially agree that the TF’s authors are the main authors of their systems; in 53% of the valid answers we receive a positive or partially positive answer regarding the estimated truck factors.

According to the surveyed developers, documentation is the most effective development practice to overcome a truck factor event, followed by the existence of an active community and automatic tests. We also comment on the main lessons we learned from the developers’ answers to our questions.

Chapter 4

Investigating Truck Factor Events

The maintenance and evolution of open source projects frequently relies on a small number of core developers. The loss of such core developers, known as a truck factor event, might be detrimental for these projects and even threaten their entire continuation. In this chapter, we adopt a mixed-methods approach to investigate truck factor events. First, we carefully select 1,932 popular GitHub projects and observe that 315 projects (16%) experienced a truck factor event; among them, 128 projects (41%) survived their most recent truck factor event, i.e., new developers assumed the project development. Next, we conduct a survey with developers that have been instrumental in project survival. This survey indicates that (i) in most cases the new maintainers were aware of the project discontinuation risks when they started to contribute; (ii) their own usage of the systems is the main motivation to contribute to projects that faced truck factor events; (iii) human and social factors played a key role when making these contributions; and (iv) lack of time and the difficulty to obtain push access to the repositories are the main barriers faced by the new truck factor developers.

4.1 Introduction

Open source software has an increasing importance in our society: 72% of GitHub survey participants report that they always seek out OSS options when finding new tools.¹ Several popular and complex applications, including operating systems and office suites, are currently available under open source licenses. Additionally, most proprietary software nowadays depends on a variety of open source frameworks and libraries: e.g., Instagram publicly acknowledges and thanks the developers responsi-

¹<http://opensource-survey.org/2017/>

ble for the open source libraries used in their site.² Notwithstanding, there is also a growing concern on the sustainability of modern open source projects, since they are usually managed by a small number of developers, without financial support [Eghbal, 2016]. OpenSSL, a cryptography library that provides secure communications with Web servers, is a remarkable example. Despite being used by two-thirds of all Web servers, the project was maintained by a single developer until 2014, when a major bug, nicknamed Heartbleed, affecting millions of sites was detected in its implementation [Durumeric et al., 2014].

An easy way to communicate and understand the dependency of a software project on key developers is the notion of Truck Factor, also known as Bus or Lottery Factor. The Truck Factor (TF) of a project is the minimal number of developers that the project depends on for its maintenance and evolution [Williams and Kessler, 2003]. Stated otherwise, if the TF developers abandon the project (after winning in the lottery or being hit by a truck) the project maintenance will be importantly affected. Recently, a number of researchers turned their eyes on the importance of studying the Truck Factor of software projects, specifically open source ones. For example, Zazworka et al. [2010] were the first to propose a heuristic to compute TFs by mining data from version repositories. Cosentino et al. [2015] worked on a tool (and novel algorithm) for the same purpose, but targeting git-based repositories. Later, Avelino et al. [2016] proposed a heuristic to estimate TFs, based on a code authorship metric (see Chapter 3). However, we still lack studies that go beyond measuring TF towards more profound understanding of what happens when influential TF developers leave the project, a situation we refer to as a *TF event*.

In this chapter, we adopt a mixed-method approach to investigate TF events aiming to better understand what happens to projects when influential TF developers depart. On the one hand, we focus on three research questions and empirically investigate a large set of GitHub projects, as will be explained in detail next. On the other hand, we qualitatively analyze the responses of TF developers who took over a project after it was abandoned (by its original TF developers) to answer our last three research questions. The remainder of this section briefly describes these research questions and lists our contributions.

Our first question is **RQ1**. *How common are TF events in open source projects?* In a way this question is a sanity check: if TF events are rare, then the risk of project maintenance and evolution being affected by such events is limited, and the concept of TF is of limited use. Second, while a TF event can be expected to affect the project

²<https://www.instagram.com/about/legal/libraries/>

maintenance and evolution, the project might overcome those difficulties. We would like to know **RQ2**. *How often open source projects survive TF events* and **RQ3**. *What are the distinguishing characteristics of the surviving projects?* Next, if a project survived a TF event, then there is one or more developers that took responsibility for further project evolution after the first group of TF developers has left. Hence, we check **RQ4**. *Do new TF developers perceive risks of project discontinuation?* and explore **RQ5**. *What motivates a developer to assume an open source project after it faces a TF event?* Finally, we investigate **RQ6**. *What project characteristics most facilitate or hamper the work of recently arrived TF developers?*

To answer these questions we first collect and curate a large dataset of 1,932 GitHub projects. We then propose a methodology to identify TF events in the commit history of these systems, using the algorithm presented in Chapter 3. We use this methodology to provide quantitative answers to the first three research questions. Finally, we report the results of a survey with 33 new TF developers, i.e., developers who assumed the maintenance of a studied project after it was abandoned by its original TF developers. We use this survey to provide qualitative answers to the last three proposed research questions.

Our contributions are threefold. *First*, we propose a methodology to identify TF events by mining software repositories and particularly to identify systems that survive these events (Section 4.2). *Second*, we show that TF events are not just a theoretical concept: we provide a throughout characterization of 357 truck factor events, detected during the evolution of 1,932 popular GitHub projects; furthermore, we show that only 41% of the projects facing a TF event fully recover the maintenance work after the events (Section 4.4). *Finally*, by surveying TF developers that assumed the maintenance of the surviving systems, we reveal that 53% were previously using these systems, i.e., they were motivated by their own need to fix bugs or to implement features; we also found that human and social characteristics had a key role to make the work of these contributors easier (Section 4.5).

We organize this chapter as follows. Section 4.2 defines concepts and methodologies used in the study. Section 4.3 provides a description of our study design. Section 4.4 presents the results of a quantitative investigation. Section 4.5 reports the results of a survey with TF developers of the surviving systems. Section 4.6 summarize and discusses the key study findings. Sections 4.7 discuss threats to validity. Section 4.9 concludes the study.

4.2 Truck Factor

In this section, we first define important concepts used in this chapter. Then, we describe the approach used to identify TF events and the systems that survived such events.

4.2.1 Definitions

These are key definitions used throughout this chapter.

- *Truck factor (TF)*: minimal number of developers on a project that have to be hit by a truck (or quit) before the project gets in serious trouble [Williams and Kessler, 2003; Zazworka et al., 2010; Lavallée and Robillard, 2015].
- *TF developers*: minimal set of developers $\{d_1, d_2, \dots, d_n\}$ that if leave will put a project in trouble. Typically, algorithms to estimate TF also compute this set.
- *TF event*: when all TF developers abandon the project, putting its maintenance and evolution in trouble.
- *Surviving system*: a system that survives a TF event, by attracting new TF developers who assume its maintenance.

4.2.2 Identifying Truck Factor Events

To search for TF events, we first estimate the TF of a system at a time t and verify whether the TF developers abandoned the system before t . In this study, we consider that a developer *abandoned* a project if her last commit occurred at least one year before the most recent repository commit. Existing studies rely on different thresholds to classify developers inactivity or departure from a project, including three months [Constantinou and Mens, 2017b], six months [Lin et al., 2017; Foucault et al., 2015], and one year [Izquierdo-Cortazar et al., 2009]. Since truck factor is viewed as a drastic event, we decide to be conservative and use a minimal one-year of inactivity period to define that a developer abandoned a project. To estimate truck factors we use the algorithm proposed in Chapter 3.

We use an example to illustrate how we identify TF events. Consider the fragment of the `composer/satis`³ development history shown in Figure 4.1. To preserve the privacy

³<https://github.com/composer/satis>

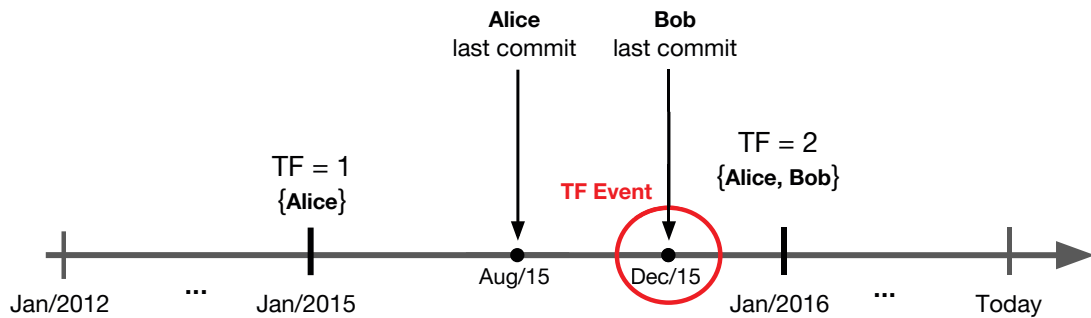


Figure 4.1. TF event on *composer/satis*

of the contributors involved in our example, we replace their usernames with fictitious ones. Suppose we first compute the system’s TF in January 2015. At this point, the TF estimated by the algorithm equals one since *Alice* is the (unique) TF developer. As *Alice* is active in January 2015 (she has a commit after this date), no TF event is observed in this first period. In the next computation (January 2016), TF increases to two, with *Alice* and *Bob* as the TF developers. Moreover, both developers abandoned the project before this date: *Alice* in August 2015 (date of her last commit) and *Bob* in December 2015. Therefore, we assume that *composer/satis* faced a TF event in December 2015, when the last TF developer abandoned the project. We repeat this procedure in intervals of one year, aiming to check for TF events in multiple points of a project history.

4.2.3 Identifying Surviving Projects

Although TF events have a major impact in the maintenance and evolution of software projects, projects can survive such events. In other words, by definition a TF event puts a project at serious risk, but it does not necessarily imply project termination. For example, after TF events the development may continue with new developers, who have taken charge of the project. In contrast, when no significant development is performed after a TF event, the project is at risk. Such projects are identified in our analysis since subsequent TF computations provide an identical set of TF developers as the ones of previous time periods.

Definition: Formally, let TF_1, TF_2, \dots, TF_n , be a sequence of TF developer sets at different time periods, where TF_t represents the TF developer set of a system S computed at time t . We say that S *survived* a TF event at t_1 if there exists t_2 , $1 \leq t_1 < t_2 \leq n$, such that

- (a) all $d \in TF_{t_1}$ abandoned the project before t_1 , and

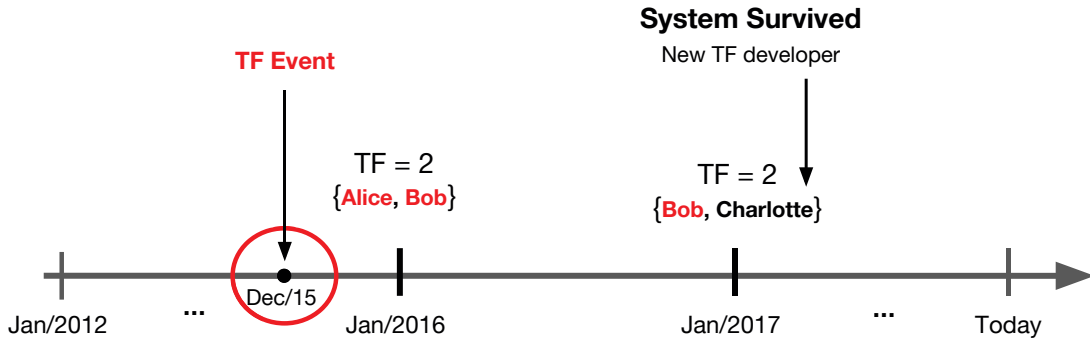


Figure 4.2. Surviving on a TF event on *composer/satis*

- (b) $TF_{t_2} \setminus TF_{t_1} \neq \emptyset$, i.e., at the time interval $[t_1, t_2]$ at least one new developer performed important contributions to the point of entering in TF_{t_2} .

As illustrated in Figure 4.2, our running example (*composer/satis*) follows these two conditions, assuming $t_1 = \text{Jan } 2016$ and $t_2 = \text{Jan } 2017$. All TF developers identified in January 2016 have abandoned the project before this date (*condition (a)*). Furthermore, in January 2017, a new TF developer (*Charlotte*) is identified (*condition (b)*). Therefore, we consider that *composer/satis* survived a TF event.

As a final note, a project may subsequently face multiple TF events. In general, a system survives if it survives all observed TF events (if any), or, equivalently, the most recently observed TF event (if any).

4.3 Study Design

We adopt a mixed-methods approach and combine a large scale analysis of version control repository data with a survey. Mixed-methods are appropriate for the pragmatic stance common in software engineering research, and were often applied in the past [Easterbrook et al., 2008].

4.3.1 Dataset & Preprocessing

To perform the quantitative part of the study, we built a dataset with GitHub projects. Initially, we focus on six programming languages with the largest number of GitHub repositories: JavaScript, Python, Ruby, C/C++, Java, and PHP. Next, we select the top-500 most starred repositories for each of those languages at the moment of analysis (June 2017). Next we cloned the selected repositories. To safeguard the quality of the

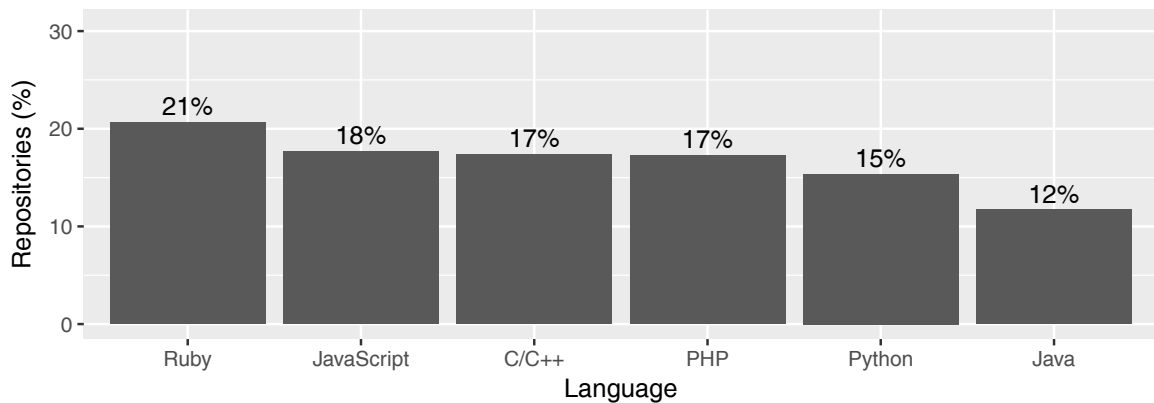


Figure 4.3. Number of projects by language.

dataset we filter the resulting collection of 3,000 GitHub repositories, as explained in the next paragraphs.

First of all, some projects do not use GitHub exclusively or did not use GitHub exclusively during their entire history. Moreover, as recognized by Kalliamvakou et al. [2015], migrating to GitHub from a different platform does not necessarily preserve the commit history. Since the identification of TF events depends on developer commit activity, partial omission of the project history might lead to misidentification of TF developers, thus threatening the validity of our results. To eliminate this threat, we filter out repositories where more than 50% of the files are added in less than 20 commits: we consider such a massive import of files an indication that an important part of the project history has taken place outside GitHub. By applying this filter, we exclude 677 projects. Second, the algorithm we use to compute TFs requires at least two years of commit history. Therefore, we filter out repositories containing less than two years of development activity; in doing so, 338 more projects are excluded.

We complement the filtering process with a manual inspection of the resulting 1,985 systems. Particularly, we manually inspect the project description searching for indications that the project does not represent a software unit. Among others, we found repositories containing books, awesome-lists (sets of suggested books, links, etc.), technology code samples, and projects that, in their description, explicitly state themselves as unmaintained. We have manually identified and excluded 53 projects. The resulting dataset is composed of 1,932 ($= 3,000 - 677 - 338 - 53$) projects.

As shown in Figure 4.3, most projects are implemented in Ruby (398 projects, 21%); on the other side, Java is the language with fewest projects (226 projects, 12%). Figure 4.4 shows violin plots with the distribution of the number of developers, files, commits and stars per project (please note the logarithmic scale). The median values

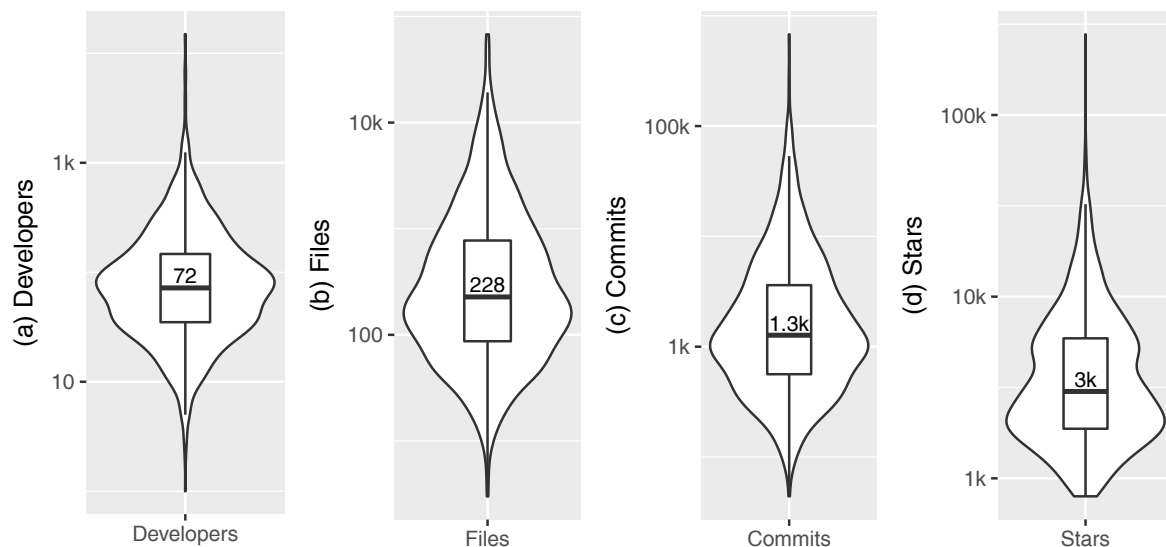


Figure 4.4. Distribution of the number of developers, commits, files, and stars.

are indicated inside the violin plots. We conclude that the dataset constructed typically includes large systems, both in size and in number of developers; the systems also have a large number of commits and are popular (number of stars).

4.3.2 Aliases Handling

The correctness of TF computations highly depends on the set of distinct developers. However, developers do not necessarily use only one alias when contributing to a project [Kouters et al., 2012; Goeminne and Mens, 2013]. Therefore, it is important to detect and handle aliases among the developers of the 1,932 projects in our dataset. Essentially, aliases occur when the same developer uses multiple e-mails to push commits to a system [Kouters et al., 2012; Wiese et al., 2016]. To detect aliases, we use a feature of the GitHub API that maps a commit author to its GitHub account. Essentially, GitHub uses the e-mail address in the commit header to link the commit to a GitHub user. Using this feature, we mapped each developer of each system to their GitHub account. In a given system, developers d_1 and d_2 are considered the same when they share the same GitHub account. As a downside, this approach does not handle the cases where developers have multiple GitHub accounts. However, in our experience and preliminary tests, this situation is rare. The most frequent cause of aliases is developers using different machines, with their local GitHub configured with distinct e-mails.

Figure 4.5 shows a violin plot with the percentage of aliases in each project. As we can observe, there is a significant percentage of aliases in our dataset (1st quartile=

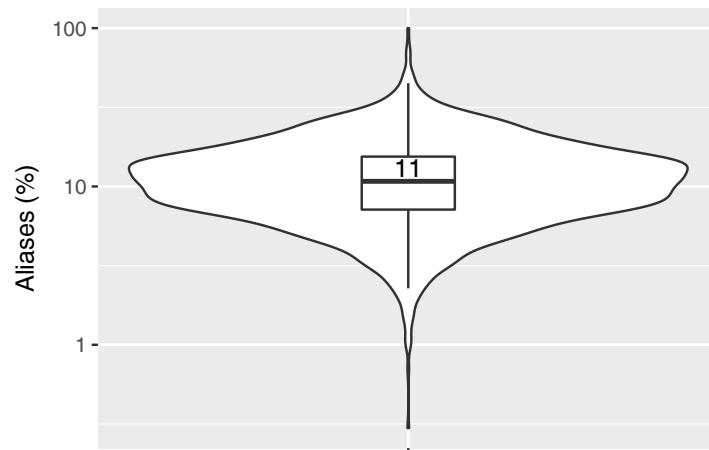


Figure 4.5. Percentage of aliases in each project

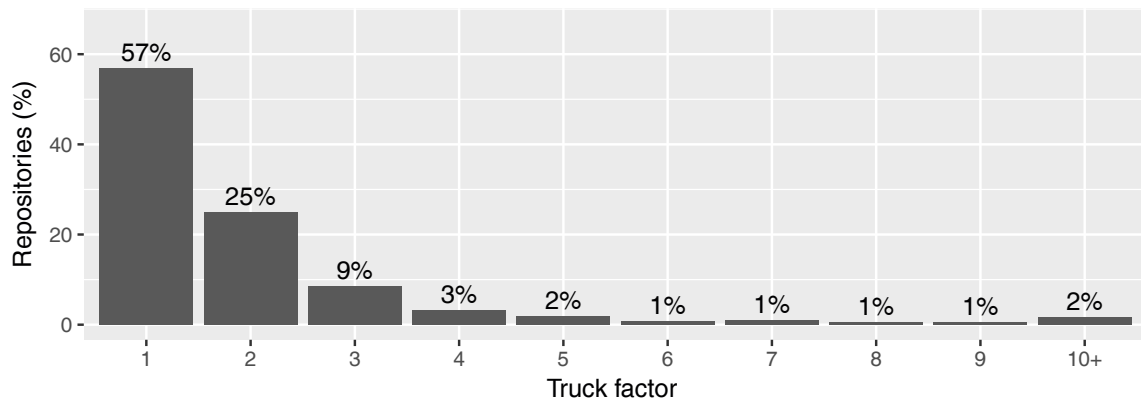


Figure 4.6. TF of the 1,932 projects in our dataset

7%, median= 11%, and 3rd quartile= 15%). `yusugomori/DeepLearning` is an interesting outlier: the system has a single developer, who made commits using five different e-mails. Therefore, the percentage of multiple aliases in the system is 100%.

4.3.3 Estimating Truck Factors

After selecting the systems and handling aliases, we compute TFs for the 1,932 projects in our dataset, using the last version in our cloned repositories. Figure 4.6 presents a histogram with the TF results. As we can observe, most projects have a low TF. For example, the percentage of projects with $TF = 1$ is 57%, while less than 6% have a TF higher than 5. The highest TF is 26, computed for `edx/edx-platform`, which is the software platform that supports edX massive open online courses. These findings concur with the earlier results of Chapter 3 that reported that 65% of the evaluated systems have

$TF \leq 2$, based on a sample of 133 popular GitHub projects.

Most open source projects have low TFs. In a sample of 1,932 projects, 57% have $TF = 1$ and 25% have $TF = 2$. The highest TF in our sample is 26 developers.

4.4 Searching for TF Events and Surviving Projects

In this section, we describe a quantitative exploration of the collected data, aiming to answer three key research questions:

RQ1. How common are truck factor events in open source projects? To start our investigation, we assess whether TF events indeed happen in open source development.

RQ2. How often open source projects survive a truck factor event? Assuming the previous question reveals that TF events indeed occur, this second question takes a step further and investigates how often projects overcome such events.

RQ3. How surviving projects differ from non-surviving ones? Finally, assuming we find projects that survived their TF events, we compare them with other projects that did not have the same fate. The goal is to identify characteristics that might help projects to overcome the loss of TF developers.

RQ1. Truck Factor Events

How common are truck factor events in open source projects?

We identify truck factor events in 315 projects, 16% of our dataset. Most of the projects faced only one event (88%); however some projects faced two (11%) or even three (< 0.1%) truck factor events. Figure 4.7 shows the percentage of TF events grouped by truck factor. As expected, most events are observed in systems with a small truck factor. For example, 66% of the truck factor events happens in projects with a truck factor equal to one. This means that most projects that face TF events are maintained by one core developer. In contrast, only two TF events occur in projects with a TF higher than four: `etsy/logster` ($TF = 7$) and `PointCloudLibrary/pcl` ($TF = 6$). `etsy/logster` is a small project, with only 13 files and 117 commits at the TF event. By contrast, `PointCloudLibrary/pcl` is a large project, with 9,568 commits and 2,204 files, when the TF event was detected. All TF developers started to contribute to this project in the first year of its development (2011). Although they were active contributors (more than

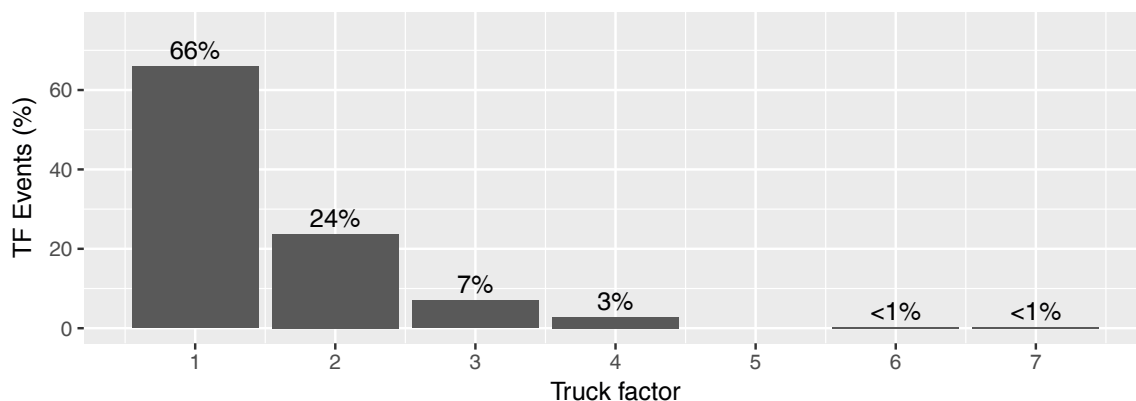


Figure 4.7. Projects facing TF events

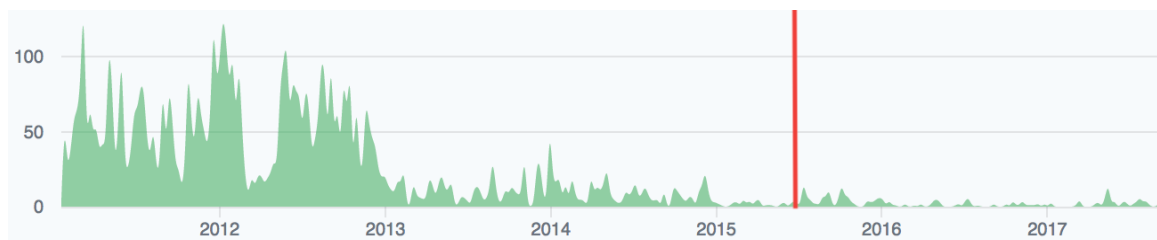


Figure 4.8. Contributions to PointCloudLibrary/pcl over time (screenshot from GitHub). A TF event occurred at June, 2015 (vertical red line), according to the proposed methodology.

90 commits), they abandoned the project before 2015. To show the impact of their departure, Figure 4.8 shows a screenshot with the contributions to PointCloudLibrary/pcl, as available on its GitHub page.⁴ We can see that most contributions happened before June, 2015, when the project faced a TF event according to our methodology (vertical red line, in the figure). This was the date of the last commit of one of the TF developers. The commits of the other five TF developers all happened before May, 2014. It is interesting that PointCloudLibrary/pcl has had financial support from a non-profit organization,⁵ as indicated in the project’s README page on GitHub. However, the site of this organization and its accounts in social networks do not receive updates since 2014, which is therefore close to the TF event date, as identified by our methodology.

Truck Factor is not just a theoretical concept: 16% of the studied projects faced at least one TF event during their development; 66% of these events happened in systems with TF=1, which are 55% of the projects.

⁴<https://github.com/PointCloudLibrary/pcl/graphs/contributors>

⁵<http://www.openperception.org>

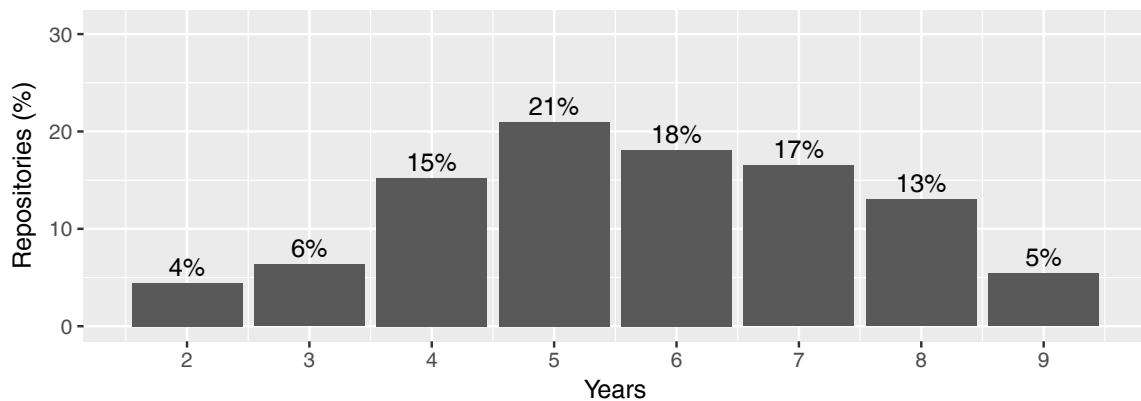


Figure 4.9. Age of the repositories with TF events

Figure 4.9 shows the age of the repositories with TF events, considering their creation date on GitHub. As we can see, most projects facing TF events (71%) have between 4 and 7 years of development. In the full dataset, 61% of the projects have between 4 and 7 years. Figure 4.10 shows when these events happen, in terms of number of development years and counting only the first event, for projects with multiple TF events. As we can observe, there is a concentration of TF events in the first years of development; 59% of the studied events happened in the first two years of development. In fact, in some cases the event happened right after the repository creation. For example, in 24 projects the TF developers abandoned the projects in the first six months.

59% of the TF events happened in the first two years of development; but 71% of the projects with TF events have now between 4 and 7 years of development.

RQ2. Survival Rate

How often open source projects survive a truck factor event?

A project survives a TF event when it survives the last observed event. In total, 128 projects (out of 315 projects) overcome their TF events, which represents a survival rate of 41%. In most cases (86%) we detected that only one TF developer was attracted to the project and was responsible for its survival. However, there are cases where two (12%) or even three (2%) new TF developers were attracted to the projects. Additionally, in 64% of these cases the attraction occurred in the first year after the

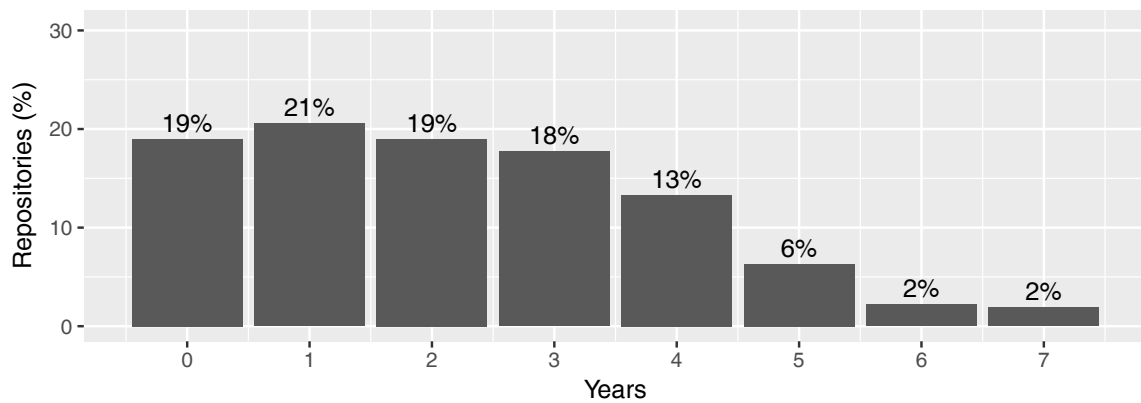


Figure 4.10. When do TF events happen (counting from the repositories creation)

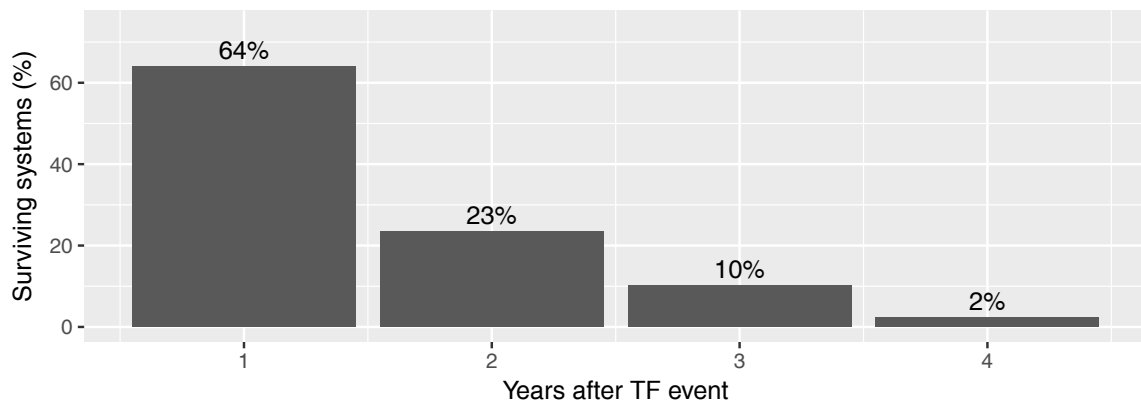


Figure 4.11. When do projects survive a TF event

TF event, as presented in Figure 4.11. Therefore, as expected, it is more difficult to recover project maintenance after years of inactivity.

It is possible to recover from TF events: 41% of the projects survived their last observed TF event, usually by attracting a single new TF developer (86%).

A developer is called a *newcomer* if her first commit occurs after the last observed TF event; otherwise, she is an *old-contributor*. In most surviving projects (52%), the new TF developers are all *old-contributors*. However, a significant part of the projects survived with the help of *newcomers* (41%) or by attracting both newcomers and old contributors (7%).

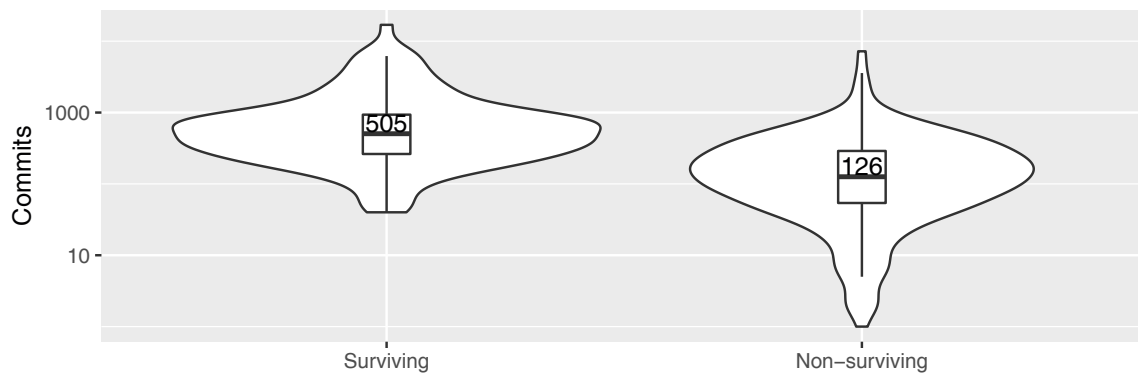


Figure 4.12. Number of commits after the last observed TF events

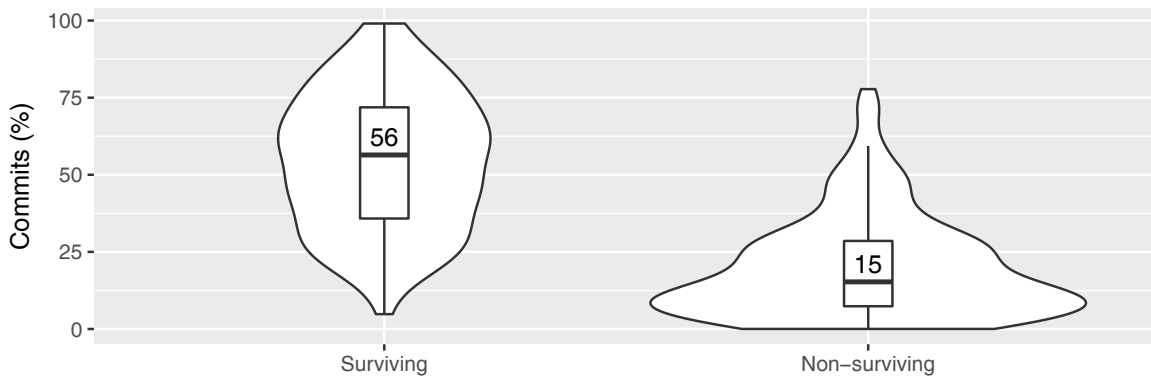


Figure 4.13. Percentage of commits after the last observed TF events

Newcomers play a key role when recovering from TF events. They participated in the recovery of 48% of the surviving projects.

RQ3. Surviving vs Non-surviving Projects

How surviving projects differ from non-surviving ones?

First, Figures 4.12 and 4.13 show respectively the distribution of the absolute number and the percentage of commits after the last TF event detected in each surviving project (128 projects) and also in the non-surviving ones (187 projects). Before discussing these figures, we stress that a TF event should have a major impact on a project maintenance and evolution, but this does not necessarily mean the project maintenance has ceased after the event. Therefore, we can have commits after TF events in non-surviving systems; however, these commits are not performed by important developers. This means that the projects continue to be at risk even in the presence of commits after

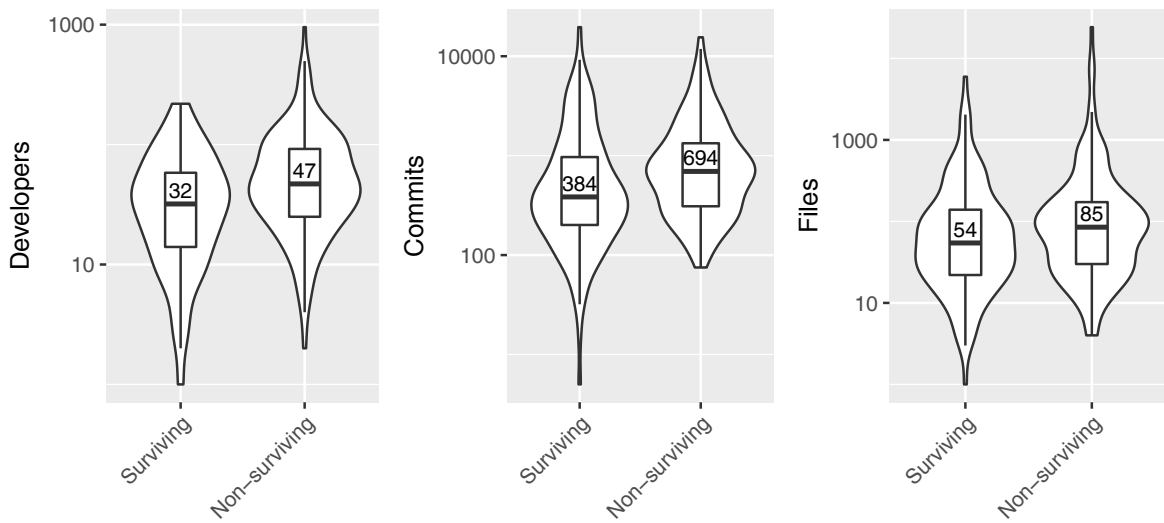


Figure 4.14. Number of developers, commits and files, for surviving and non-surviving projects (at the date of the studied TF events)

the TF event. Indeed, the violin plots in Figures 4.12 and 4.13 show a clear difference between surviving and non-surviving systems. The surviving systems have 505 commits (56%) after the last detected TF event, whereas the non-surviving ones have only 126 commits (15%), considering the median values. The third quartile measures are 949 commits (72%), for surviving projects; and 289 commits (29%), for non-surviving ones. These differences are confirmed using the one-sided version of the Mann-Whitney test ($p\text{-value} \leq 5\%$). The effect size, according to Cliff's delta [J. Grissom and J. Kim, 2005], is *large* in both cases: $d = 0.64$, for the number of commits; and $d = 0.79$, for the percentage of commits after the last TF events.

After confirming the difference between the relative number of commits after the TF events of the surviving and non-surviving projects, we also compare them using other metrics. Figure 4.14 shows violin plots with the distribution of the number of developers, number of commits, and number of files of the surviving and non-surviving projects. All values refer to the date of the studied TF events. Interestingly, the surviving projects have less developers than the non-surviving ones (32 vs 47, median values). They also have less commits (384 vs 694, median values) and less files (54 vs 85), which is confirmed using the one-sided version of the Mann-Whitney test ($p\text{-value} \leq 5\%$). However, as computed using Cliff's delta, the effect size of this difference is *negligible* for number of files ($d = 0.13$) and *small* for number of commits ($d = 0.25$) and developers ($d = 0.26$).

At the moment of the TF events, we found no major difference between surviving and non-surviving projects, in terms of number of developers, commits, and files.

4.5 Survey with TF Developers

In this section, we report the results of a survey with the new TF developers of the surviving projects, i.e., developers that played a major role in the maintenance of these systems after the identified TF events. We rely on this survey to provide answers to the last three research questions:

RQ4. Do new TF developers perceive risks of project discontinuation? The intention is to check whether the developers perceived the projects being at risk, before making the contributions that led them to reach a TF developer status.

RQ5. What motivates a developer to assume an open source project after it faces a TF event? In this investigation, we consider that a developer assumes a project when it becomes one of its TF developers. The intention is to provide insights on how open source project managers should proceed to attract new TF developers to their systems.

RQ6. What project characteristics most facilitate or hamper the work of recently arrived TF developers? The intention is to shed light on programming and management practices that should be promoted (or avoided) in open source development.

4.5.1 Survey Design

For each surviving project (128 projects), we select their *new* TF developers. After excluding the ones without a public and valid e-mail address, we have identified 144 potential participants. We sent an e-mail to these developers with four questions: (1) *Did you think that [project] was at risk of being discontinued before deciding to make major contributions to its continuation?* (2) *Why did you decide to make these contributions?* (3) *What project characteristics and practices helped you to make these contributions?* (4) *What were the main barriers you faced when making these contributions?*

From the 144 e-mails we sent, four returned due to an invalid address. In total, we received 33 answers, representing a response rate of 24% ($33/(144 - 4)$); this rate is more than what has been achieved by previous studies [Palomba et al., 2015; Vasilescu et al., 2015]. To process the answers, we rely on Thematic Analysis [Cruzes and Dyba,

2011], which consists of (i) initial reading of the developer responses; (ii) generating initial codes for each response; (iii) searching for themes; (iv) reviewing the themes to find opportunities for merging among themes; and (v) defining and naming the final themes. These steps were performed by the author of this thesis; after that the final themes were checked and revised with a second member of our research group.

RQ4. Discontinuation Risks

Do new TF developers perceive risks of project discontinuation?

We started the survey with the question about the perception of the discontinuation risks, according to the new TF developers. The purpose of this question is twofold. First of all, it serves as an additional validation of the importance of the TF event: if developers believe that the project is at risk of discontinuation after the TF event, then further evolution of the project is indeed threatened. Second, this question assesses awareness of the new TF developers; awareness of the context and therefore of their tasks helps team coordination in software development [Espinosa et al., 2007].

Table 4.1 summarizes the results for this question. Most respondents (18 developers, 60%) agreed that the projects were facing risks of discontinuation. Examples of positive answers include: *“when I thought the project would die, I started making contributions to it once again”* (D8); and *“yes, otherwise the project would have been completely abandoned”* (D9). Furthermore, we classify as a partial agreement five cases (17%) where the developers reported problems in the projects, but were not clear about their severity, or mentioned the problem was mitigated by another developer who also entered in the TF set. As examples, we have these answers: *“[the] development had slowed”* (D14); and *“A new primary developer stepped in and took responsibility of the project after the original developer left.”* (D32). Indeed, the developer mentioned by D32 was also identified as a new TF developer in our study; therefore, we contacted him for our survey, but unfortunately he did not reply to our e-mail. Finally, six developers (20%) answered that they did not perceive the projects as being at risk. Usually, these developers were succinct in their answers (just answering *no*, for example). Remarkably, among the negative answers, one developer mentioned the project is supported by a major software company, which contributes to reduce the discontinuation risks, in his opinion: *“this open source project is actually backed by a for profit company, so the project didn’t risk being abandoned”* (D24). Finally, we classify one answer as unclear, because it is not related to the provided question. Four respondents did not answer this survey question.

Table 4.1. Did you perceive the projects at risk?

Yes	Partially	No	Unclear
18 (60%)	5 (17%)	6 (20%)	1 (3%)

Table 4.2. Motivations to contribute

Motivations	Devs	%
Because I was using the project	17	53
To contribute to an open source project	11	34
To avoid the project discontinuation	5	16
I have interest on the project area	4	13
I get paid to contribute	4	13
To improve my own skills	3	9
I have the skills required by this project	3	9
It is a successful project	3	9
Others	7	22

77% of the TF developers were aware (or partially aware) of the risks faced by the surviving systems, before making the contributions responsible for the project recovery.

RQ5. Motivations

What motivates a developer to assume an open source project after it faces a TF event?

With this question, we aim to reveal the reasons that motivated the new TF developers to make their major contributions to the projects. Table 4.2 summarizes the main reasons mentioned by the surveyed developers. **Because I was using the project** to address my personal or professional needs is the most common reason, according to 17 participants (53%). As examples, we have these answers: *"I used the [project] in my own products and was struggling with a few bugs so I decided to fix them and contribute back"* (D14); *"mostly because I used [project] heavily and was asked for documentation and improvements from within my company"* (D16); and *"I used the project professionally and was in a position to provide some level of support as part of my job"* (D23). **To contribute to an open source project** is the second most common reason, according to 11 participants (34%). For example, one participant mentioned *"contributing to open-source to give back to community"* (D4). **To avoid the project discontinuation** is mentioned by five developers (16%), as in these answers: *"I was the only additional contributor on the project so I was the only person capable of keeping it alive"* (D7); and

Table 4.3. Characteristics that helped new TF developers

Type	Characteristics	Devs	%
Human/ Social	Friendly and active owners/members	12	41
	I liked/knew the project	3	10
Technical	Programming language	4	14
	Well-known SE principles	4	14
	Pull based development	4	14
	Continuous integration	2	7
	Clean and well-designed code	2	7
	Code revision	1	3
Others	Main repository access	3	10
	Job support	2	7
	Small or simple project	2	7
	Open source license	1	3

"it will die if I don't [contribute], and I think it still has value." (D13). The remaining reasons mentioned by the participants are as follows: *I have interest on the project area* (4 developers), *I get paid to contribute* (4 developers), *to improve my own skills* (3 developers), *I have the right skills to contribute to this project* (3 developers), *it is a successful project* (3 answers), *it is a promising or interesting project* (2 answers), *I was invited to contribute* (2 answers), *for personal satisfaction* (2 answers), and *to attract developers to my company* (1 answer). One respondent did not answer this question.

The developers responsible to reactivate the maintenance of the surviving projects were motivated by their own usage of the projects (17 developers, 53%). They also intended to contribute back to an open source community (34%) or avoid the project discontinuation (16%).

RQ6. Enablers and Barriers

What project characteristics most facilitate or hamper the work of recently arrived TF developers?

We start with the project characteristics that facilitated the attraction of the new TF developers. As presented in Table 4.3, we organize these characteristics in three groups: human and social characteristics (15 answers), technical characteristics (17 answers), and other characteristics (8 answers). The most mentioned human and social characteristic is the presence of friendly and active project owners or members (12 answers).

Table 4.4. Barriers faced by new TF developers

Type	Barriers	Devs	%
Human/ Social	Lack of time	7	26
	Lack of experience	3	11
	Unfriendly maintainers	2	7
Technical	Need to keep backward compatibility	4	15
	Lack of well-known SE principles	1	4
Others	Lack of access to the main repository	5	19
	Large number of pending issues	3	11
	No financial support	2	7
No barriers	-	4	15

As examples, we have these answers: “*it has been [dev-name]’s kindness to my first contributions and his help to me, and later other cool developers’ support*” (D6); “*the responsiveness of the existing maintainer was the key factor to my ongoing contributions*” (D11). Among others, technical characteristics include the usage of a specific programming language (4 answers) or following well-known software engineering principles and practices (4 answers). The last category groups factors like permission to access the main repository (3 answers) and financial support by a company (2 answers). Four respondents did not answer this question.

The characteristics that helped on the attraction of new TF developers have a social, technical or external nature. Friendly and active maintainers is the most mentioned facilitator, indicated by 12 developers (41%).

To complement the answer to RQ6, we also asked the new TF developers about the barriers they faced when making the contributions that led them to achieve a status of TF developer. As in the case of the first part of the question, we organize the answers mentioned by the participants in three groups: human and social barriers (12 answers), technical barriers (5 answers), and other barriers (10 answers). Table 4.4 presents the answers in each group. As we can observe in this table, most answers denote human and social barriers. Particularly, *lack of time* is the most common barrier mentioned by the survey participants (7 answers). As examples, we have these answers: “*I have other projects to maintain.*” (D3); and “*time is always an issue, especially because the range of features is fairly wide*” (D23). Technical barriers include the requirement to *keep backward compatibility and do not introduce bugs* (4 answers) and the lack of solid software engineering principles (1 answer). Another barrier commonly mentioned by

the participants is the *difficulty to obtain access to the main repository* (5 answers). The participants justify the need to obtain this access because the *maintainers are absent* (D2, D12) or *the project was abandoned* (D8, D9). Four developers mentioned they faced no barriers at all. Six developers did not answer this question.

Human and social barriers are the most common ones faced by new TF developers; particularly, lack of time is the most common barrier.

4.6 Discussion

In this section, we summarize and discuss the relevance of our study findings.

Truck factor is not only a theoretical metaphor: In the case of open source development, it is possible to argue that truck factor is just a theoretical scenario, since the code is public and others can assume the maintenance work if the key developers abandon the project. In fact, one of the participants of the survey provides an argumentation in this direction: *“it’s open source, if people want to use it, they will use it. If it’s missing features they really want/need, they will submit PR’s, or fork and maintain their own copy.”* (D30). Undoubtedly, if the code is public on GitHub, anyone has the legal permission (according to the project’s open source license) to collaborate with or fork the project. Moreover, GitHub provides many useful instruments to facilitate this process, like easy forking or pull requests. Despite that, our study shows that even popular projects may fail to attract new contributors after being abandoned by the original TF developers. More precisely, only 41% of the projects have fully recovered the maintenance activity after the TF events studied in our work. We hypothesize that assuming the maintenance of an open source project is a complex task, which requires time, technical and social skills and familiarity with the project domain; many projects therefore do not succeed to find developers with this profile and face serious maintenance problems or even fail after being abandoned by their TF developers.

Interestingly, we also found arguments in the opposite direction, stating that Truck Factor is a less important concern in software projects backed by a company regardless of the project being open source. We received at least two answers hinting in this direction, as this one: *“Most of the questions are not relevant because [Project] is actually a large project with formal sponsorship by [Company]”* (D33). In other words, these developers consider that Truck Factor is a real concern only in projects without financial support, as is the case of most open source projects.

How to overcome a TF event: Although we show that TF events are a reality in open source development, we also found that it is possible to survive these events and to recover the maintenance after attracting new developers to the TF set. By surveying these new TF developers, we shed light on two key characteristics of the surviving projects. *First*, the surveyed developers decided to assume the maintenance of these projects motivated by their own needs, since they were using the projects and require new features or fix existing bugs. Therefore, this finding suggests a connection between the number of users of an open source project and its resilience to TF events. Particularly, 53% of the TF developers surveyed in our study were attracted because they were earlier users of the projects and therefore had personal interests in avoiding their failure. *Second*, human and social factors have a key role on attracting new TF developers. According to the survey participants, 51% of the factors that helped in their attraction are social in nature; and 44% of the barriers faced in this process are also human and social ones. The importance of human and social barriers to technical contributions was also observed by Palomba et al. [2018]. Our findings, therefore, confirm the importance of human and social factors in open source development. This is particularly the case if most contributions are voluntary, as indicated by one of the survey respondents: *“There is no authority over the top that chooses who will work on what. We are all contributing during our “free” time, for only the “enjoyment” of it. So, we sort of contribute only where it “feels” good.”* (D26)

4.7 Threats to validity

External Validity: The dataset used in this study was carefully selected from popular projects on GitHub, coming from six different programming languages. However, our findings cannot be generalized to other projects and particularly to closed-source projects. Indeed, our survey results suggest that TF events in the context of software with financial support might have very different characteristics.

Internal Validity: Our approach uses data from the entire development history of a project to identify TF events, therefore it is sensitive to the loss of parts of this history (e.g., migrating the project’s code from another version control system to GitHub without preserving the previous development history). To mitigate this threat, we remove from our dataset the projects with evidence of a corrupted migration to GitHub—see Section 4.3.1. We also manually removed non-software projects, such as books and tutorials.

Construct Validity: With respect to construct validity, our results depend on the accuracy of truck factor computations. Therefore, to mitigate this threat we used the TF algorithm that presents the best accuracy, as pointed by a recent comparative study [Ferreira et al., 2017]. Another threat to the validity of our work stems from the selected threshold of one year to identify abandoners. However, there is no consensus in the literature on appropriate thresholds to identify such developers, e.g., Constantinou and Mens [2017a] used a 1-year threshold as well, while Lin et al. [2017] used a 180-day threshold to find developers abandoning a project. Since this study focuses on key developers, we believe that a less strict threshold can reduce the effect of this threat in our analyses. Additionally, the truck factor measures may also vary due to possible developer aliases. We mitigate such a threat by carefully handling the most common sources of aliasing—see Section 4.3.2.

4.8 Related work

Truck factor is a concept defined by the agile community to assess knowledge concentration in software projects. As the concept initially lacked a formal definition, the first works in this area focused on proposing algorithms to compute truck factors. The first algorithm to this purpose was proposed by Zazworka et al. [2010]. After that, it was used by Ricca and Marchetto [2010] and Torchiano et al. [2011], respectively, to investigate the presence of “heroes” in open source projects and to investigate threshold values to use when computing truck factors. However, as further demonstrated by Ricca et al. [2011] and Hannebauer and Gruhn [2014], Zazworka’s algorithm suffers from scalability problems, which limits its applicability to real systems. To address these problems, new algorithms were proposed: Cosentino et al. [2015] proposed a hierarchical algorithm, which aggregates file-level authorship results to modules and, in a second step, aggregates module-level results into systems; Rigby et al. [2016] proposed a solution inspired by a Monte Carlo simulation algorithm; and Avelino et al. [2016] use code authorship metrics to identify source code files’ authors [Fritz et al., 2010, 2014]. Essentially, Avelino’s algorithm relies on a greedy approach to identify the developers that together control the authorship of most files in a system. In a recent work, Ferreira et al. [2017] compared these three algorithms and concluded that Avelino’s algorithm is the most accurate one. However, the aforementioned works did not investigate whether TF events really occur and what happens with open source projects after such events.

Truck Factor can be considered as a particular case of turnover, involving the principal developers of a project. Turnover of developers in general is a well-studied

phenomenon in software engineering. Foucault et al. [2015] report the negative impacts of turnover in the internal quality of five open source projects. Hilton and Begel [2018] recently studied internal turnover in a major software company, with more than 30K employees. By surveying a sample of 374 of such employees, they reveal what causes engineers to consider leaving their teams, why they leave, how they learn about new teams, and how they decide which team to join. Lin et al. [2017] conducted a similar study, but with focus on five open source projects. They show that developers are retained when they (i) start contributing to the projects earlier, (ii) maintain both code developed by others and their own code, and (iii) mainly code instead of writing documentation.

Motivations and barriers to contribute to open source systems were previously investigated for different developer profiles: one-time code contributors (developers that have only one patch accepted) [Lee et al., 2017], casual contributors (developers with few contributions and who have no intention to become an active project member) [Pinto et al., 2016], newcomers [Steinmacher et al., 2015, 2016], and core developers [Coelho et al., 2018]. Regarding the reasons that motivate developers to contribute to open source, some of these studies also show that core developers are motivated by their personal needs, as we concluded for the specific case of new TF developers. By contrast, one-time contributors and casual contributors are mainly motivated by the need to fix minor bugs. Lack of time is a common barrier to contribute, mentioned by core developers, one-time contributors, and casual contributors. It was also commented by the new TF developers surveyed in our study. Steinmacher et al. [2015] defined a conceptual model composed of 58 barriers that may hamper newcomers' first contributions. They list and classify these barriers, but do not provide insights on which are the most common ones. Coelho and Valente [2017], report a survey with the maintainers of 104 failed open source projects, i.e., projects that are not maintained anymore. According to their survey, the most common reasons for open source project failures are the appearance of a strong competitor, obsolescence, and lack of time or interest of the project owners.

4.9 Conclusion

In this chapter, we presented an in-depth investigation of the occurrence of TF events in open source projects, i.e., the abandonment of a project by its principal developers. We showed that TF events are not only a metaphor, but they indeed happen in open source projects (in 16% of such projects, at least in our sample of 1,932 GitHub projects).

Additionally, we showed that projects survive such events, by attracting new core contributors (41% of the projects survived a TF event, in our sample). Finally, we reveal the motivations that led these developers to take over the studied projects, after the projects faced a TF event; we also reveal the principal enablers and barriers faced by these developers during this process. This list of enablers and barriers can be used by project leaders to improve the management practices employed in their projects.

Finally, open source communities should be made aware of successful cases of projects overcoming TF events, as we report in our study, and motivate developers to actively contribute to projects at risk due to TF events.

As a final note, our data is publicly available at this GitHub repository: https://github.com/gavelino/tfevents_data.

Chapter 5

Identifying Software Maintainers

In large and complex systems, identifying developers capable of maintaining a source code file is an important but challenging task. In this context, repository-mining techniques can help by providing some level of automation. Still, whether such techniques effectively identify skilled software maintainers is yet unclear. To shed light on this issue, we evaluate three techniques supporting software maintainers recommendation, namely (1) the number of changes a developer makes; (2) the number of lines a developer owns in the last version of a file; and (3) a linear regression approach for defining experts. We apply these techniques against the evolution history of ten systems, contrasting recommendations with an oracle built from surveying developers. We concluded that practitioners should use the approach based on linear regressions, since it has the best performance after controlling for size/recency, closely followed by number of commits.

5.1 Introduction

When software needs to be fixed or improved, identifying who is able to maintain, assist, or review a particular source code file can be a wicked task. Particularly, many software projects follow collective code ownership practices, as encouraged for example by agile software development methodologies. In these projects, it is common to have files with multiple contributors, sometimes reaching dozens of them. Among these contributors, some are responsible for the major changes in the files, while others can be considered as peripheral contributors, who perform only minor and less important maintenance tasks. However, usually, there is no clear and easily defined frontier separating peripheral from major contributors [Joblin et al., 2017]. This separation is important for example when a critical bug is reported and project managers have to

rapidly identify an expert developer capable to fix it.

In this context, different repository-mining techniques can be applied to identify skilled software developers from the historical data kept by version control systems (VCS) [McDonald and Ackerman, 2000; Mockus and Herbsleb, 2002b; Fritz et al., 2014]. By mining the development history, the goal is to infer developers able to maintain a file or a more specific unit of code. However, the extent that existing techniques successfully recommend software maintainers, as well as possible limitations, remains unclear. We shed light on this matter by evaluating three popular techniques: (1) the Number of Commits [Bird et al., 2011; Casalnuovo et al., 2015]; (2) the Number of Lines of Code in the Last Version Girba et al. [2005]; Rahman and Devanbu [2011] and; (3) the Degree of Authorship (DOA) [Fritz et al., 2014]—a linear regression approach for defining experts. We apply all three against the evolution history of 10 systems (2 commercial and 8 open-source), comparing results with an oracle we build from surveying developers. We point out for the use of recency and file size information as a strategy to improve the effectiveness of the compared techniques. We also recommend the use of the DOA technique to identify maintainers, since it has the best performance, after controlling for size/recency, closely followed by *Commits*.

We organize this chapter as follows. Section 5.2 presents the three techniques evaluated in this study. Section 5.3 provides a description of our study design. Section 5.4 presents the results of the techniques comparison which are better examined and discussed in Section 5.5. Finally, Sections 5.6 and 5.7 examine threats to validity and concludes the study, respectively.

5.2 Evaluated Techniques

From the existing literature, we find three main techniques to recommend expertise from version control systems. The selected techniques depend on measures that are available on git-based version control systems or that are straightforward to compute. Moreover, although the selected techniques do not represent an exhaustive list of approaches to recommend software maintainers, they cover the key concepts adopted by most of them.

Number of Changes (*Commit*)

This technique counts the number of changes to define the experts on a file. The expertise of a developer d over a file f is defined by counting the number of commits

performed by d in f . Bird et al. [2011] and Casalnuovo et al. [2015] use it to identify experts at the level of modules and methods, respectively.

Number of Lines of Code in the Last Version (*Blame*)

This technique relies on blame-like tools to obtain the developer who last modified a line in a file. It considers expertise as the percentage of the number of lines associated to a given developer. Girba et al. [2005] consider a file expert the developer with more associated blame lines in a given system snapshot. Rahman and Devanbu [2011] rely on this technique to assess expertise of developers responsible for defective code.

Degree of Authorship (DOA)

As proposed by Fritz et al. [2014], this technique considers three events to compute the degree of authorship (DOA) of a developer d in a file f : first authorship (FA), number of deliveries (DL), and number of acceptances (AC). If d is the creator of f , FA is 1; otherwise it is 0; DL is the number of changes in f made by d ; and AC is the number of changes in f made by other developers. The DOA measure is defined as follow:

$$DOA(d, f) = 3.293 + 1.098 * FA + 0.164 * DL - 0.321 * \ln(1 + AC) \quad (5.1)$$

In this equation, FA and DL contribute to increment the DOA value, the former with higher importance than the latter. In an opposite way, changes made by other developers (AC) decrease the DOA value. The weights used in this equation were empirically derived from a study with Java developers [Fritz et al., 2014]. Although their use in other systems is as threat, the authors of the DOA metric claim the model is robust enough to be applied in different systems. Additionally, we previously used DOA to compute the truck factor of popular systems on GitHub, obtaining positive feedback from developers (Chapter 3).

5.3 Study Design

5.3.1 Target Systems

We compare the described techniques in ten systems, including two commercial systems and eight open-source systems. The commercial systems (Commercial #1 and Commercial #2) are, respectively, a web platform for digital media and the client of a VoIP

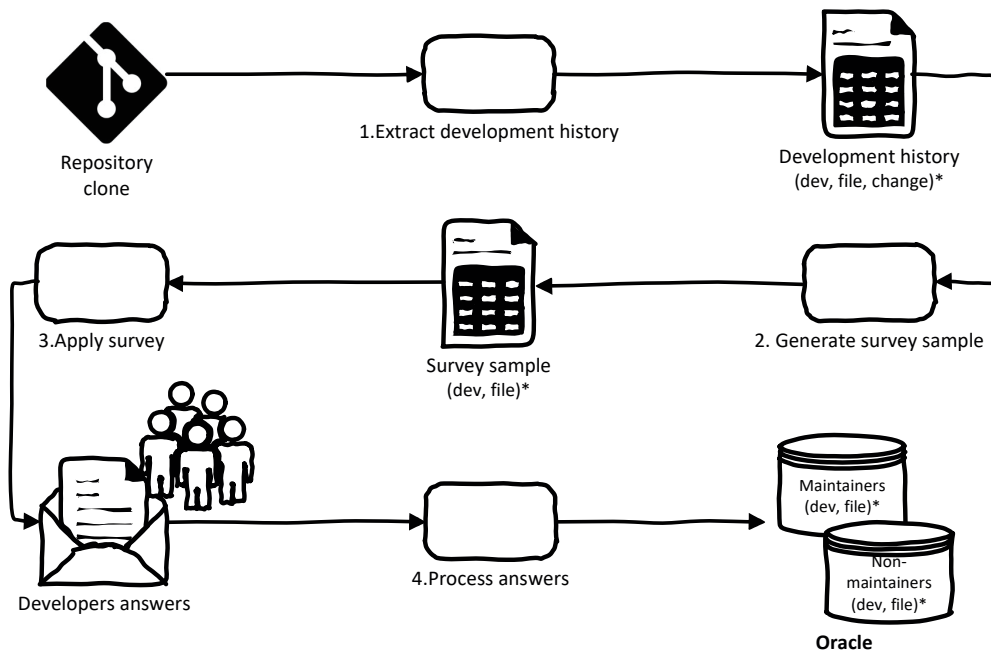


Figure 5.1. Steps to construct the authorship oracle

communication system. We omitted the names due to confidential reasons. They are developed by different teams and represent the main product of two companies, which are located in Brazil (Commercial #1) and Canada (Commercial #2). We also used the following open-source systems: Salt, Django, Moment, Ember.js, Faker, Monolog, Fog, and Puppet. These systems, which are implemented in four different programming languages, come from the truck factor study described in the Chapter 3.

5.3.2 Oracle

Our investigation requires an oracle to compare the results produced by the techniques. We create this oracle by asking the system developers about their knowledge on a random set of files. As illustrated in Figure 5.1, the oracle construction comprises four steps:

Step 1: Extract development history. We extract the development history from the repositories using the `git log --no-merges --find-renames` command, which returns all no-merge commits of a repository and identifies possible file renames. We process each commit, extracting three pieces of information: (i) the file path; (ii) the developer who performed the change; and (iii) the type of the change—addition, modification, or rename. Rename information is used to join the development history of a file (old and new file names). Additionally, we discard files that do not contain source code

(e.g., images, documentation) and third-party libraries. We also handle developers alias. To perform these tasks we follow the same automatic procedures described in Section 3.3.1.

Step 2: Generate survey sample. Given the development history of a system, we first discard developers with invalid e-mails and source code files that are touched by only one developer. The survey sample for a given system is generated executing the following procedure: (i) we randomly select a file and retrieve the list of developers who changed it; (ii) we discard the file if at least one of these developers reached the maximal limit of files (*files_limit*); (iii) otherwise, we add the file to the list of each developer; (iv) we repeat the procedure until there are no more files to be verified. After consulting the commercial systems' managers, we decide to ask each developer on her knowledge about a list of at most 50 files (*files_limit* = 50). For the open-source systems, we set *files_limit* = 10 to do not discourage the developers to answer the survey. This second step produces a list of pairs (*developer*, *file*) for each system in our dataset. In total, we generate a sample with 3,068 pairs, covering 1,109 files and 740 developers.

Step 3: Apply the survey. After producing the survey sample, we send an e-mail to each developer *d* asking him/her to assess his/her knowledge on each file *f* in the sample. The developers are invited to rank their knowledge using a scale from 1 (one) to 5 (five), where (1) means no knowledge about the file; and (5) means complete knowledge about the file. We also request them to explain or comment their answers in an optional text field. In total, we send 668 e-mails and received answers from 159 developers, resulting in a response rate of 24%. Additionally, these answers correspond to 1,209 pairs (*developer*, *file*), covering 654 files.

Step 4: Process answers. Figure 5.2 (Answers) shows the distribution of the answers we received for each group of systems. Score five is the most popular one in the commercial systems (31%), while score three is the most popular in the open-source systems (27%). We applied the non-parametric Wilcoxon-Mann-Whitney test [Wilcoxon, 1992] to compare the two distributions and the result shows they are statistically different (*p-value* < 0.01). Finally, we classify the answers in two disjointed sets: *declared maintainers* (O_M) and *declared non-maintainers* ($O_{\bar{M}}$). A *declared maintainer* is a developer who declared to have a knowledge greater than three in a file; otherwise, she is a *declared non-maintainer*. As we can observe in Figure 5.2 (Type), most developers are *declared non-maintainers* (52% and 54%, for commercial and open-source systems, respectively). Therefore, although all the respondents had changed the files included in the study at least once, most of them answered they have limited

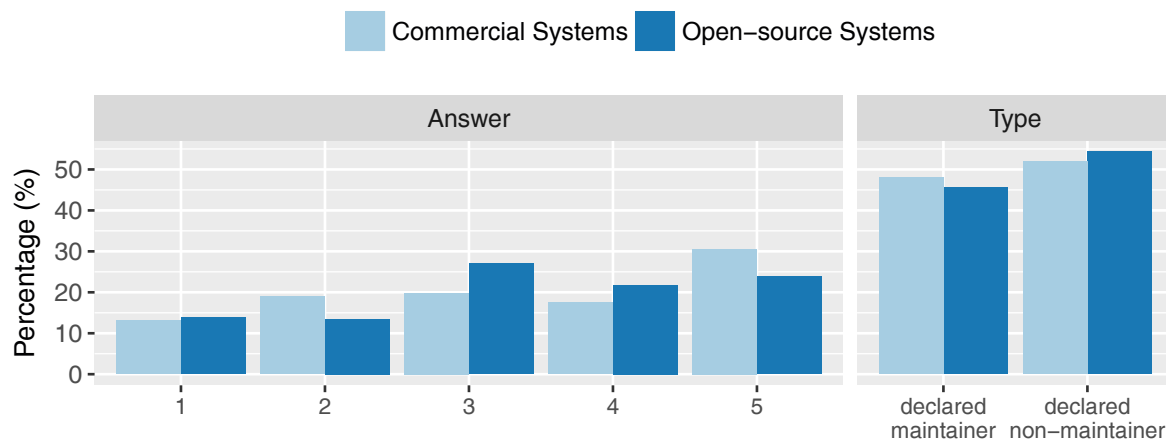


Figure 5.2. Distribution of the survey answers

knowledge on these files. The data generated for this study is publicly available at: <https://github.com/gavelino/authorship-data>.

5.3.3 Inferring Maintainers and Non-maintainers

In this section, we describe how we use the techniques presented in Section 5.2 to classify the developers as *maintainers* or *non-maintainers* of a file. First, for a given file f we normalize the measures produced by each technique. We define that $expertise(d, f)$ is 1 to the developer with the highest measure for f , otherwise it receives a proportional value. For example, if f_1 was modified by three developers d_1 , d_2 , and d_3 and they performed, respectively, five, four, and two commits, their normalized value using the *Commit* technique are $expertise(d_1, f_1) = 1.0$, $expertise(d_2, f_1) = \frac{4}{5} = 0.8$, and $expertise(d_3, f_1) = \frac{2}{5} = 0.4$. A similar normalization happens with the *Blame* and DOA measures (i.e., the highest measure is normalized to 1; the other measures receive a proportional value in the range 0 to 1).

We consider the developer d of a file f as a *maintainer candidate* if she has an *expertise* greater than or equal to a threshold k ; otherwise she is a *non-maintainer candidate*. For instance, by taking the previous example, if we adopt $k = 0.5$, d_1 and d_2 are classified as *maintainer candidate* of f_1 , while d_3 is classified as *non-maintainer candidate*. For the sake of clarity and brevity, *maintainer* and *non-maintainer* candidates are just called *maintainer* and *non-maintainer* in the remainder of this article.

5.3.4 Evaluation Metrics

Although in the survey we obtained a response ratio greater than the one common in software engineering studies [Kitchenham and Pfleeger, 2008; Palomba et al., 2015;

Vasilescu et al., 2015], our oracle is not complete, because not all developers who changed a file answered the survey. Therefore, it is not possible to compare the techniques using precision and recall because these measures require a complete ground truth, which can be used to answer whether any recommendation is correct or not. Instead, using the oracle data, we calculate the ratio of correct classifications produced by each technique (*Commit*, *Blame*, and *DOA*). Specifically, we compute the *maintainers hit ratio* (HR_M) of a given technique using the following equation:

$$HR_M(k) = \frac{|\{(d, f) \in O_M \mid expertise(d, f) \geq k\}|}{|O_M|} \quad (5.2)$$

Therefore, $HR_M(k)$ is the ratio of *declared maintainers* (O_M) that a given technique correctly identifies. As described in Section 5.3.3, we consider that d is a *maintainer* of a file f if $expertise(d, f) \geq k$. The *hit ratio of non-maintainers* ($HR_{\overline{M}}$) is computed using a similar approach, but using the set $O_{\overline{M}}$ and considering as *non-maintainers* the ones whose $expertise(d, f) < k$, as follows:

$$HR_{\overline{M}}(k) = \frac{|\{(d, f) \in O_{\overline{M}} \mid expertise(d, f) < k\}|}{|O_{\overline{M}}|} \quad (5.3)$$

Both HR_M and $HR_{\overline{M}}$ are important to evaluate the results of the studied techniques. For example, a high HR_M but a low $HR_{\overline{M}}$ may indicate the technique is inflating the number of *maintainers*, erroneously classifying many developers with low knowledge as *maintainers*. Therefore, we also compute the *harmonic mean* (HM) of the hit ratios of *maintainers* and *non-maintainers*, as given by the following equation.

$$HM(k) = \frac{2 * HR_M(k) * HR_{\overline{M}}(k)}{HR_M(k) + HR_{\overline{M}}(k)} \quad (5.4)$$

We adopt an *harmonic mean* instead of the *arithmetic mean* because the former is less sensitive to outliers. For example, suppose that $HR_M = 90\%$ and $HR_{\overline{M}} = 20\%$. This is not an interesting result, because $HR_{\overline{M}}$ is very low. In this case, HM is 33%, whereas the arithmetic mean is 55%. Therefore, the *harmonic mean* better reflects (and penalizes) the unbalanced nature of these HR_M and $HR_{\overline{M}}$ values.

To illustrate the use of these evaluation metrics, Figure 5.3 presents a toy example. The figure shows developers d_1 , d_2 , d_3 , and d_4 who changed a given file f and their *expertise* measures as computed by *Blame*, *Commit*, and *DOA*. We can also observe the sets of *declared maintainers* ($O_M = \{d_2, d_4\}$) and *declared non-maintainers* ($O_{\overline{M}} = \{d_1, d_3\}$) for f . In the bottom, we present the classification results produced by the proposed evaluation metrics, assuming $k = 0.5$. *Blame* correctly classifies one *maintainer* (d_2) and one *non-maintainer* (d_1). Since $|O_M| = |O_{\overline{M}}| = 2$, then

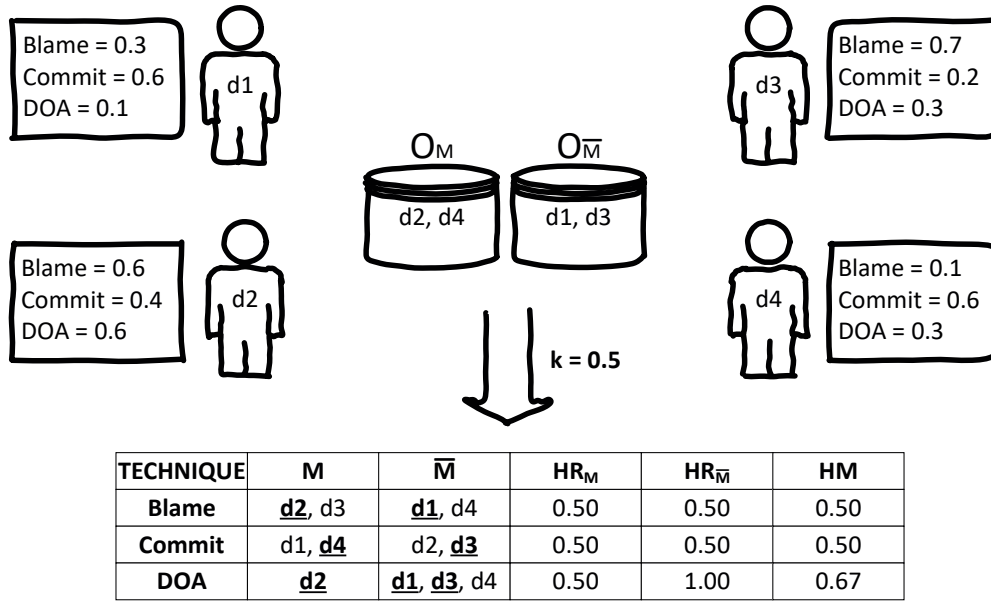


Figure 5.3. Computing HR_M , $HR_{\bar{M}}$, and HM for a hypothetical file f . In the bottom part, underlined values denote developers correctly classified by the techniques as *maintainers* (M) and *non-maintainers* (\bar{M}) according to the oracles O_M and $O_{\bar{M}}$, and assuming a threshold $k = 0.5$.

$HR_M = HR_{\bar{M}} = \frac{1}{2} = 0.50$, and $HM = 0.50$ for *Blame*. The same results are obtained by *Commit*. DOA correctly classifies one *maintainer* (d_2) and two *non-maintainers* (d_1, d_3), obtaining the following results: $HR_M = \frac{1}{2} = 0.50$, $HR_{\bar{M}} = \frac{2}{2} = 1.00$ and $HM = 0.67$.

5.4 Results

To compare the techniques to infer *maintainers* and *non-maintainers*, we use the *harmonic mean* (HM), as described in the previous section. As this measure depends on a threshold k , we vary k from 0 to 1, using steps of 0.1. By considering the results in Figure 5.4, we can identify the best technique for each group of systems (commercial and open-source). For commercial systems, *Blame* obtains the highest HM (67%, $k = 0.1$), closely followed by *Commit* (66%, $k = 0.2$), while DOA has the lowest one (63%, $k = 0.6$). On the other hand, for open-source systems the best result is obtained by *Commit* (63%, $k = 0.6$), closely followed by DOA (62%, $k = 0.8$) and *Blame* (60%, $k = 0.1$). In summary, although *Blame* and *Commit* provide slightly better results, respectively for commercial and open-source systems, the three techniques present similar performance on classifying developers as *maintainers* or *non-maintainers* (HM ranging from 60% to 67%).

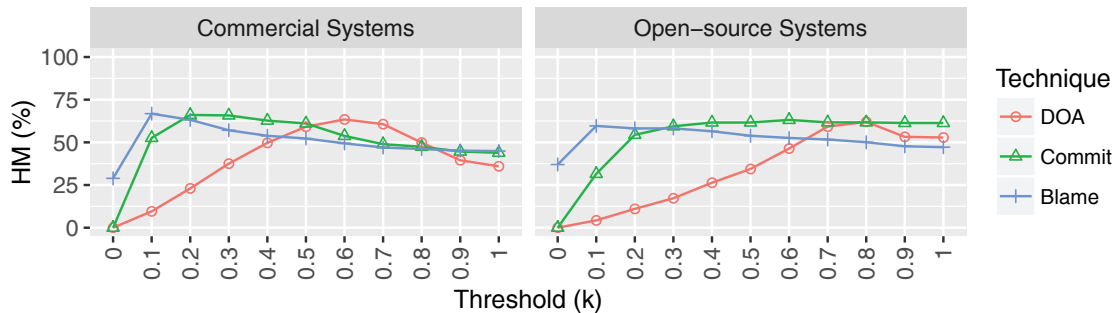


Figure 5.4. Harmonic mean (HM)

In Figure 5.4, we also observe that DOA is more susceptible to threshold variations, while *Commit* and *Blame* provide more stable results, specially when we consider $k \geq 0.2$. Additionally, *Commit* and mainly *Blame* achieve high HM measures ($> 50\%$) with a low k , while DOA requires $k > 0.5$ to produce values higher than 50%.

5.5 Discussion

In this section, we analyze examples in which the studied techniques fail and we also shed light on their limitations. Then, we evaluate the impact of using different thresholds to judge the technique results in scenarios where they are more likely to fail.

5.5.1 When and Why the Techniques Fail

We start by contrasting the cases where the three techniques succeed (*AllHit*) on identifying the *maintainers* of the studied systems against the cases where they all fail (*AllMiss*). To compute these cases we configured the techniques with their best thresholds, as pointed in Section 5.4. We identified 270 pairs (*developer, file*) in *AllHit* and 96 pairs in *AllMiss*. For each pair (*developer, file*) in *AllHit* or *AllMiss*, Figure 5.5 shows the distribution of the percentage of commits performed in the selected *file* by the respective *developer*. We can see that when all the techniques fail (first plot) the percentage of commits by the considered developers is usually lower than when they all succeed (second plot). Indeed, in most of the *AllMiss* cases the percentage of commits by the selected developers is low—75% of the missed *maintainers* have less than 8% of the files’ commits. In other words, in the *AllMiss* cases the studied techniques failed because the developers classified themselves as *maintainers* despite having a small percentage of the files’ commits. To clarify the failure reasons, we manually inspected the 96 pairs (*developer, file*) in *AllMiss*. We found three major reasons:

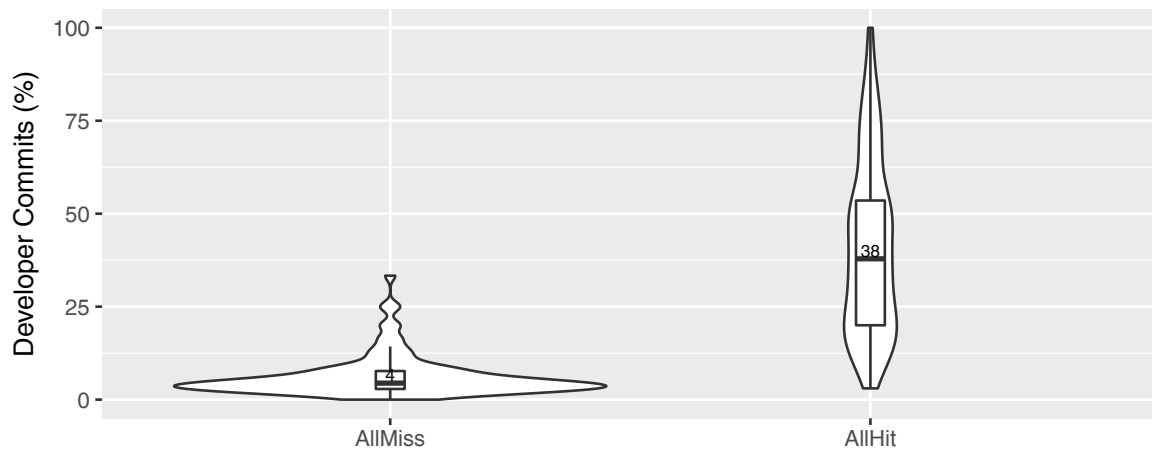


Figure 5.5. Distribution of the percentage of commits by a developer d who is a declared maintainer of a file f when all techniques failed to identify this condition (*AllMiss*) and when all techniques correctly identified this condition (*AllHit*).

- Recency:** in 24 cases (25%), there is less than one month that the developers modified the file. Although most of them had done few and minor contributions, they considered to have good knowledge on these files. Furthermore, *recency* influences other 11 cases (11%) where the developer is the last one to change the file. However, the *recency* of the contributions is not caught by any of the studied techniques. For example, a concern about this question is mentioned in the following answer of a Django developer: “*I don’t know if you are taking time into account, but I’d expect this to be a significant factor*”. We also found evidences that the familiarity with a file decays over time. For example, we received the following comment of a Puppet developer: “*I believe I submitted a patch about 3 years. At the time, I probably understood what I was doing but it’s too long ago now*”.
- File size:** the number of lines of a file also seems to influence the results. In fact, in 13 failures (14%) the changed files are small (≤ 26 lines of code, which is the first quartile of the *file size* distribution in the entire oracle). This factor is mentioned by a Django developer to justify his answer: “*This is a tiny file, so having added one line of code, I’ve contributed a significant portion of it I guess*”. By contrast, a large file requires more effort to gain knowledge on its content, as exposed by another Django maintainer: “*On a relatively big piece (such as this 1K+ lines file) with a bunch of authors, I might be very knowledgeable about the piece of code I touched but know basically nothing about the rest*”.

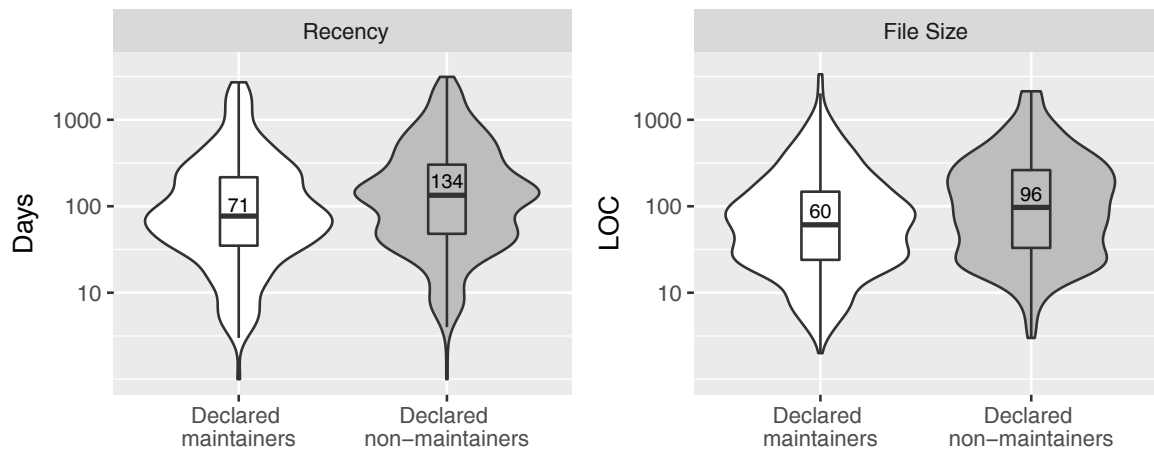


Figure 5.6. Distribution of *declared maintainers* and *declared non-maintainers* according to the following factors: *recency* (in days from the last developer commit) and *file size* (in LOC).

- **Extra-repository activities:** in the remaining cases, we could not find evidences from the collected data to support the failure results. For example, in Commercial #2, a developer rated his knowledge with a score four but performed only one commit, months ago, in a highly modified and large file (344 commits, by 25 different developers, 700 lines of code). Analyzing this case, we found that he recently worked in a new client-side module, and the file in question is a facade widely used by this new module. This suggests that activities not reflected in the commit history can also be a source of code knowledge. Additionally, two Commercial #1 developers mentioned their participation in the design of the system as the source of knowledge in specific files (e.g., “*this file represents a concept defined in the beginning of the project. The knowledge came from participating in the file definition*”).

To better evaluate the failure reasons identified in the manual investigation, Figure 5.6 presents the distribution of the pairs (*developer, file*) regarding *recency* (in days from the last commit) and *file size* (in LOCs). The entire oracle is represented, but we show separately *declared maintainers* and *declared non-maintainers* pairs. According to a Wilcoxon-Mann-Whitney test, the presented distributions are statistically different ($p\text{-value} < 10^{-8}$, in both cases). Regarding *recency*, the median number of days from the last commit is 71 days for *declared maintainers* and 134 days for *declared non-maintainers*. Regarding *file size*, the median size is 60 LOC for *declared maintainers* and 96 LOCs for *declared non-maintainers*. This result reinforces that developers are

Algorithm 2: ISMAINTAINER TEST

Input: Developer d , file f
Output: **True** if d is a maintainer of f ; **False**, otherwise

```

1 function isMaintainer ( $d, f$ )
2   begin
3      $adj \leftarrow 1$ ;
4     if " $f$  is a small file" then
5        $adj \leftarrow adj - k_1$ ;
6     else if " $f$  is a large file" then
7        $adj \leftarrow adj + k_2$ ;
8     if " $d$  recently modified  $f$ " then
9        $adj \leftarrow adj - k_3$ ;
10    else if " $d$  modified  $f$  a long time" then
11       $adj \leftarrow adj + k_4$ ;
12     $adjThreshold \leftarrow k * adj$ ;
13    return  $expertise(d, f) \geq adjThreshold$ ;
14  end

```

more likely to declare themselves as *maintainers* when they recently modified a file or when this file is small. By contrast, large files and files modified a long time ago tend to encourage a negative response from developers.

5.5.2 Controlling for File Size and Recency

The results presented so far depend on a threshold to classify a developer d as a *maintainer* of a file f . Different thresholds are used for each technique; but for a given technique, the same threshold is used to classify developers as *maintainers* or *non-maintainers* of all files. However, as concluded in Section 5.5.1, the techniques tend to fail when the modified files are small (or large); and when the last modification by a developer in a file was performed recently (or a long time ago). Therefore, we decided to experiment different thresholds for the mentioned scenarios. Instead of having a single threshold, we adjust this value according to the following thresholds: small files (k_1), large files (k_2), files recently modified by the developer (k_3), files modified a long time ago (k_4); plus the previously used threshold for the remaining cases (k). We used the first quartile of the *file size* distribution in lines of code to classify the small files in a system; the last quartile is used to classify the large files. The first quartile of the number of days of the last commit by the developers of a system is used to classify the recently modified files; the last quartile classifies the files modified a long time ago. For each system, we experiment different values of k_i , ranging from 0.0 to 1.0 with steps of 0.05. The test described in the Algorithm 2 is then used to decide whether a developer d is a maintainer of a file f .

The improvements on the *HM* values (harmonic mean of the hit ratio of *main-*

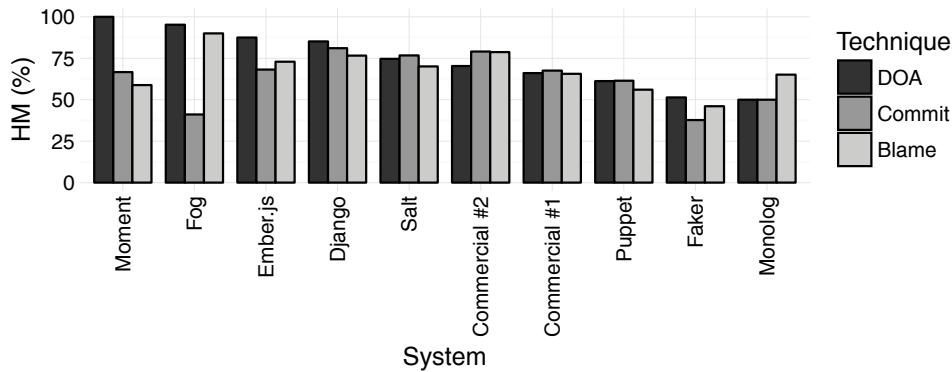


Figure 5.7. Harmonic mean (HM) of the hit ratio of *maintainers* and *non-maintainers* achieved when controlling for file size and recency

ainers and *non-maintainers*) using these thresholds are as follows:

- For *Blame*, the HM improvements range from 0% (Ember.js) to 14% (Puppet).
- For *Commit*, the improvements range from 0% (Monolog, Salt, Moment, and Django) to 41% (Fog).
- For DOA, the improvements range from 0% (Monolog and Salt) to 95% (Fog). Fog is the system with the highest improvement because most of the answers for the system refer to recently modified files.

Figure 5.7 shows the HM results of each system considering the proposed improvements. As we can see, DOA presents the highest gain after controlling for size and recency. It achieves the best HM values for six systems (one shared with *Commit*), followed by *Commit* (four systems) and *Blame* (two systems, one shared with *Commit*).

5.6 Threats to Validity

To construct the oracle we split the answers in two sets: *declared maintainers* ($score > 3$) and *declared non-maintainers* ($score \leq 3$). Although a score three may represent an acceptable knowledge, we followed a more conservative criterion, only classifying as *declared maintainers* the developers that informed a higher knowledge on the files. A second threat relates to the fact that some developers might provide unreliable answers. For example, developers might overestimate their scores aiming to obtain personal credits. To minimize this threat, we informed in the beginning of the

survey that the study has not a commercial purpose and we also avoid to send a large number of questions to the developers. Finally, the study results are based on the data extracted from 10 real-world systems. Therefore, our findings may not generalize to other systems.

5.7 Conclusion

We summarize our major findings as follows:

1. When used without control for particular cases, DOA, *Commit*, and *Blame* have similar performance, with the harmonic mean (*HM*) of *maintainers* and *non-maintainers* hit ratios ranging from 60% to 67%.
2. However, when controlling for *file size* and *recency*, the improvements on *HM* are relevant. These improvements reach 95% (DOA), 41% (*Commit*), and 14% (*Blame*).
3. After controlling for *file size* and *recency*, DOA presents the highest *HM* results in five systems, followed by *Commit* (four systems) and *Blame* (two systems).

Practical Implications: In many contexts, project managers have to identify possible maintainers for source code files. In this chapter, we investigated three techniques for this purpose, based on data extracted from version control systems. As a first practical implication, we showed that practitioners with interest in these techniques should consider file size and recency information. As a second implication, we showed that practitioners should use the DOA technique to identify maintainers, since it has the best performance (after controlling for size/recency, closely followed by *Commits*).

Chapter 6

Conclusion

This chapter concludes this thesis by revisiting each question introduced in Chapter 1 and discussing our main contributions, then pointing directions for future research.

6.1 Contributions

This thesis explored, through a set of quantitative and qualitative studies, the use of code authorship algorithms to understand the organization of system's development teams. Through the thesis we proposed new metrics, conducted empirical investigations, and surveyed systems developers. This study was guided by the following questions:

Q1. How is authorship organized and how does it evolve in software systems?

To address this question, in Chapter 2, we conducted a study where we investigated how authorship measures evolve in Linux kernel development and after that we extended the analysis to a dataset of 114 systems. We proposed a set of authorship-center metrics and used them to investigate authorship characteristics of the Linux kernel development, such as distribution of the number files per author, work specialization, and co-authorship patterns. After, we contrasted the Linux results against the ones we computed for the extended dataset. The main contributions of this study are: *(i)* an in-depth investigation of authorship in a large, successful, and long-lived open-source community, backed up by several authorship measures; *(ii)* a formal definition of several authorship-centric concepts, such as authors and specialists/generalists, which others may use as a common ground to study the social organization of software systems; and *(iii)* the confirmation that the authorship patterns observed in the Linux are also

followed by other popular open source systems.

Q2. Can code authorship metrics be used to estimate truck factors?

We addressed this question, in Chapter 3, by proposing a new approach to estimate truck factors, based on code authorship to identify key developers. We applied this approach to estimate the TF of 133 popular project systems and validated the results by surveying important developers of these systems. In summary, our contributions are: (i) a new approach for estimating truck factors, as well as empirical evidence on the reliability of its results; (ii) a tool for estimating truck factors, which is publicly available on GitHub;¹ (iii) the estimation of the truck factor of 133 popular GitHub systems, which shown that most systems have low TF (65% of the systems have a $TF \leq 2$); and (iv) a list of practices the developers see as most effective to deal with the loss of key developers.

Q3. How common are truck factor events in open source projects?

To address this question, in Chapter 4, we extended the previous TF study (Chapter 3), by investigating the occurrence of TF events in a large dataset composed of 1,932 open source systems. Our results showed that 16% of the projects faced at least one TF event, and a significant portion (41% of these projects) were able to survive the detected events. The main contributions of this study are: (i) a methodology to identify TF events and surviving systems based on mining repository data; (ii) an empirical confirmation that truck factor is not a theoretical concept, by identifying a representative number of TF events in an open source dataset; and (iii) a list of motivations, enablers, and barriers that developers face when helping open source projects to overcome a TF event, collected by surveying these developers.

Q4. Can code authorship identify software maintainers?

To address this question, in Chapter 5, we selected three well-known authorship techniques and compared their performance on identifying source code maintainers in 10 systems (2 commercial and 8 open source systems). To evaluate the techniques performance we relied on an oracle built by surveying the systems' developers. Our contributions are: (i) an evaluation of the effectiveness of code authorship techniques on suggesting code maintainers; (ii) an analysis of the limitations of the techniques which provides insights on how to improve them by controlling code size and recency data; and (iii) an oracle of code expertise with 1,209 pairs (*developer, file*), which can be

¹<https://github.com/aserg-ufmg/Truck-Factor>

used in other studies, since it is publicly available on GitHub.²

6.2 Discussion

In this section, we discuss and put our findings in perspective.

6.2.1 Code Authorship Applicability

Code authorship has been used to identify and recommend experts for parts of the code [McDonald and Ackerman, 2000; Mockus and Herbsleb, 2002a; Minto and Murphy, 2007; Begel et al., 2010], however its application to investigate how the work force of a software system is organized and evolves was not explored before. In the studies developed in this thesis, we showed that code authorship can be used to identify important characteristics of development teams, including potential risks, such as knowledge concentration. Additionally, as code authorship is easily computed from data available on source code repositories, the measures proposed in this thesis can be used by supporting tools to monitor and control important properties and practices in software development. For example, they can be used to identify the ability of an open source system to attract not only minor contributors, but also key developers; to check whether the criteria used to decompose a system in modules is indeed able to foster the specialization of the work force, as usually expected from software modularization [Parnas, 1972]; to check practices like collective ownership of the code base, as commonly advocated by agile methodologies [Beck and Andres, 2004]; and to assess the concentration of knowledge in few team members. In that direction, we developed a prototype tool that provides, among other information, authorship measures about popular systems on GitHub. This tool is publicly available at <http://gittrends.io>.

6.2.2 Truck Factor is a Real Risk in Open Source

Additionally to propose a new approach to estimate truck factors (Chapter 3), we also investigate cases where truck factor events indeed occur (Chapter 4). By analyzing the development history of a large set of open source projects we were able to identify a significant number of TF events. We also showed that although some of these projects were able to overcome the detected TF events, this is not the case for most of them. These results show that TF events is really a risk. Specially, because many popular projects have a small truck factor. Although we only investigated TF events in open

²<https://github.com/gavelino/authorship-data>

source projects, we cannot discard the potential risks of TF events in commercial projects, once developer turnover is also an issue in these systems [Zhou and Mockus, 2010; Rigby et al., 2016; Hilton and Begel, 2018].

Interestingly, by comparing the answers given by the developers in the two TF studies (Chapters 3 and 4), we observed divergences among the project characteristics cited as most relevant to attenuate the loss of TF developers. In the first study, the developers mentioned documentation as the most important project characteristic, while in the second study, the presence of a friendly and active community was the most common answer. While technical factors are important and they are also mentioned by the new TF developers, human factors may be more relevant when helping projects to survive TF events. This divergence seems to indicate a discrepancy about what is perceived as important and what is really important when helping projects to deal with the loss of important developers.

6.3 Future Work

The authorship investigations conducted to address **Q1**, **Q2**, and **Q3** used datasets built with open source systems, collected from GitHub. Although we took care to select a large sample of important and representative projects, our findings cannot be generalized to other projects and particularly to closed-source projects. Therefore, future research should investigate to which extent our results are applicable to such systems. In special, they can investigate whether trends such as skewed distribution of the number of files per authors, high number of specialists, and low Truck factor, also happen in commercial systems. To facilitate replication, we publish our tools and datasets on GitHub. They are available at https://github.com/gavelino/phd_thesis.

Among the answers we received when addressing **Q2** and **Q3**, there are suggestions on how improving our metrics. Future research can introduce and evaluate these improvements suggested by the surveyed developers. As an example, to consider the relative importance of the code units (e.g., by counting the number of other code units that depend on them) in the estimation of truck factors. Additionally, the study conducted to address **Q4** pointed improvements opportunities on code authorship techniques, by controlling data about file size and recency of the changes. These improvements can provide more accurate code maintainers suggestions and also can help to improve the results proposed in this work, including TF estimations.

When addressing **Q3**, we focused our analysis on the surviving projects, by sur-

veying the developers that helped their projects to survive the TF events. Future work can investigate the non-surviving projects looking for reasons why they were not able to survive.

Finally, another line of future work is the design, implementation, and evaluation of tools to assess the risks faced by an open source project, in case it is abandoned by its TF developers. For example, to inform which part of the project will become unmaintained in such cases. This assessment is particularly important to the users of such projects. We also see space to investigate recommenders of TF developers for a system, based for example on their own usage of the projects.

Bibliography

- Albert, R. and Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47--97.
- Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 361--370.
- Aspray, W., Mayadas, F., and Vardi, M. Y. (2006). Globalization and Offshoring of Software: A Report of the ACM Job Migration Task Force. *Report of the ACM Job Migration Task Force, Association for Computing Machinery*.
- Avelino, G., Passos, L., Hora, A. C., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1--10.
- Baldwin, C. Y. and Clark, K. B. (1999). *Design rules: the power of modularity*. MIT Press.
- Barr, E. T., Bird, C., Rigby, P. C., Hindle, A., German, D. M., and Devanbu, P. (2012). Cohesive and isolated development with branches. In *Lecture Notes in Computer Science*, volume 7212, pages 316--331. Springer.
- Beck, K. and Andres, C. (2004). *Extreme programming explained: embrace change*. Addison-Wesley, 2nd edition.
- Begel, A., Khoo, Y. P., and Zimmermann, T. (2010). Codebook: discovering and exploiting relationships in software repositories. In *32nd International Conference on Software Engineering (ICSE)*, pages 125--134.
- Bettenburg, N., Hassan, A. E., Adams, B., and German, D. M. (2015). Management of community contributions. *Empirical Software Engineering*, 20(1):252--289.

- Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don't touch my code! In *19th International Symposium on Foundations of Software Engineering (FSE)*, pages 4--14.
- Bird, C., Pattison, D., D'Souza, R., Filkov, V., and Devanbu, P. (2008). Latent social structure in open source projects. In *16th International Symposium on Foundations of Software Engineering (FSE)*, pages 24--35.
- Bowman, I. T., Holt, R. C., and Brewster, N. V. (1999). Linux as a case study: Its extracted software architecture. In *21st International Conference on Software Engineering (ICSE)*, pages 555--563. ACM.
- Casalnuovo, C., Devanbu, P., Oliveira, A., Filkov, V., and Ray, B. (2015). Assert use in GitHub projects. In *37th International Conference on Software Engineering (ICSE)*, pages 755--766.
- Cataldo, M., Kwan, I., and Damian, D. (2012). Conway's law revisited: The evidence for a task-based perspective. *IEEE Software*, 29:90--93.
- Chacon, S. and Straub, B. (2014). *Pro Git*. Expert's voice in software development. Apress, 2nd edition.
- Coelho, J. and Valente, M. T. (2017). Why modern open source projects fail. In *25th International Symposium on the Foundations of Software Engineering (FSE)*, pages 186--196.
- Coelho, J., Valente, M. T., Silva, L. L., and Hora, A. (2018). Why We Engage in FLOSS: Answers from Core Developers. In *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*.
- Constantinou, E. and Mens, T. (2017a). An empirical comparison of developer retention in the rubygems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13(2-3):101--115.
- Constantinou, E. and Mens, T. (2017b). Socio-technical evolution of the Ruby ecosystem in GitHub. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 34--44.
- Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28--31.
- Corbet, J., Kroah-Hartman, G., and McPherson, A. (2013). Who writes Linux: Linux kernel development: how fast it is going, who is doing it, what they are doing, and who is sponsoring it. Technical report, Linux Foundation.

- Corbet, J., Kroah-Hartman, G., and McPherson, A. (2015). Who writes Linux: Linux kernel development: how fast it is going, who is doing it, what they are doing, and who is sponsoring it. Technical report, Linux Foundation.
- Corbet, J., Rubini, A., and Kroah-Hartman, G. (2005). *Linux device drivers*. O'Reilly, 3rd edition.
- Cosentino, V., Izquierdo, J. L. C., and Cabot, J. (2015). Assessing the bus factor of Git repositories. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499--503.
- Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. *First Monday*, 10(2).
- Cruzes, D. S. and Dyba, T. (2011). Recommended Steps for Thematic Synthesis in Software Engineering. In *5th Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275--284.
- Dinh-Trong, T. T. and Bieman, J. M. (2005). The freebsd project: A replication case study of open source development. *IEEE Transactions on Software Engineering*, 31:481--494.
- Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., and Halderman, J. A. (2014). The matter of heartbleed. In *14th Conference on Internet Measurement Conference (IMC)*, pages 475--488.
- Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. (2008). Selecting Empirical Methods for Software Engineering Research. In Shull, F., Singer, J., and Sjøberg, D. I. K., editors, *Guide to Advanced Empirical Software Engineering*, pages 285--311. Springer London.
- Eghbal, N. (2016). Roads and bridges: The unseen labor behind our digital infrastructure. Technical report, Ford Foundation.
- Espinosa, J. A., Slaughter, S., Kraut, R. E., and Herbsleb, J. D. (2007). Team knowledge and coordination in geographically distributed software development. *Journal of Management Information Systems*, 24:135--169.
- Ferreira, M., Valente, M. T., and Ferreira, K. (2017). A comparison of three algorithms for computing truck factors. In *25th International Conference on Program Comprehension (ICPC)*, pages 207--217.

- Foucault, M., Palyart, M., Blanc, X., Murphy, G. C., and Falleri, J.-R. (2015). Impact of developer turnover on quality in open-source software. In *10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 829--841.
- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- Fritz, T., Murphy, G. C., Murphy-Hill, E., Ou, J., and Hill, E. (2014). Degree-of-knowledge: modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology*, 23(2):14:1--14:42.
- Fritz, T., Ou, J., Murphy, G. C., and Murphy-Hill, E. (2010). A degree-of-knowledge model to capture source code familiarity. In *32nd International Conference on Software Engineering (ICSE)*, pages 385--394.
- German, D. M., Adams, B., and Hassan, A. E. (2015). Continuously mining distributed version control systems: an empirical study of how Linux uses Git. *Empirical Software Engineering*.
- Gini, C. (1921). Measurement of inequality of incomes. *The Economic Journal*, 31(121):124--126.
- Girba, T., Kuhn, A., Seeberger, M., and Ducasse, S. (2005). How developers drive software evolution. In *8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 113--122.
- Godfrey and Qiang Tu (2000). Evolution in open source software: a case study. In *International Conference on Software Maintenance (ICSM)*, pages 131--142. IEEE Comput. Soc. Press.
- Goeminne, M. and Mens, T. (2011). Evidence for the pareto principle in open source software activity. In *1st International workshop on Model Driven Software Maintenance and 5th International Workshop on Software Quality and Maintainability*, pages 74--82.
- Goeminne, M. and Mens, T. (2013). A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971--986.
- Gousios, G., Pinzger, M., and Deursen, A. V. (2014). An exploratory study of the pull-based software development model. In *36th International conference on Software engineering (ICSE)*, pages 345--355.

- Greiler, M., Herzig, K., and Czerwonka, J. (2015). Code ownership and software quality: A replication study. In *12th Working Conference on Mining Software Repositories (MSR)*, pages 2--12.
- Hannebauer, C. and Gruhn, V. (2014). Algorithmic complexity of the truck factor calculation. In *Product-Focused Software Process Improvement*, volume 8892, pages 119--133. Springer.
- Hattori, L. and Lanza, M. (2009). Mining the history of synchronous changes to refine code ownership. In *6th International Working Conference on Mining Software Repositories (MSR)*, pages 141--150.
- Herbsleb, J., Mockus, A., Finholt, T., and Grinter, R. (2001). An empirical study of global software development: distance and speed. In *23rd International Conference on Software Engineering (ICSE)*, pages 81--90.
- Herbsleb, J. D. (2007). Global software engineering: the future of socio-technical coordination. In *2007 Future of Software Engineering (FOSE)*, pages 188--198.
- Hilton, M. and Begel, A. (2018). A study of the organizational dynamics of software teams. In *40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 1--10.
- Hong, Q., Kim, S., Cheung, S. C., and Bird, C. (2011). Understanding a developer social network and its evolution. In *27th International Conference on Software Maintenance (ICSM)*, pages 323--332.
- Hubert, M. and Vandervieren, E. (2008). An adjusted boxplot for skewed distributions. *Computational Statistics and Data Analysis*, 52(12):5186--5201.
- Izquierdo-Cortazar, D., Robles, G., Ortega, F., and Gonzalez-Barahona, J. M. (2009). Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *42nd Hawaii International Conference on System Sciences*, pages 1--10.
- J. Grissom, R. and J. Kim, J. (2005). *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates. ISBN 0805850147.
- Jensen, C. and Scacchi, W. (2007). Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study. In *29th International Conference on Software Engineering (ICSE)*, pages 364--374.

- Jermakovics, A., Sillitti, A., and Succi, G. (2011). Mining and visualizing developer networks from version control systems. In *4th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 24--31.
- Jiménez, M., Piattini, M., and Vizcaíno, A. (2009). Challenges and improvements in distributed software development: A systematic review. *Advances in Software Engineering*, pages 3:1--3:16.
- Joblin, M., Apel, S., Hunsen, C., and Mauerer, W. (2017). Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics. In *39th International Conference on Software Engineering (ICSE)*, pages 164--174.
- Joblin, M., Mauerer, W., Apel, S., Siegmund, J., and Riehle, D. (2015). From developer networks to verified communities: a fine-grained approach. In *37th International Conference on Software Engineering (ICSE)*, pages 563--573.
- Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 92--101.
- Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2015). An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering*.
- Kitchenham, B. A. and Pfleeger, S. L. (2008). Personal opinion surveys. In Shull, F., Singer, J., and Sjøberg, D. I. K., editors, *Guide to Advanced Empirical Software Engineering*, chapter 3, pages 63--92. Springer-Verlag.
- Koch, S. and Schneider, G. (2002). Effort, co-operation and co-ordination in an open source software project: Gnome. *Information Systems Journal*, 12:27--42.
- Kouters, E., Vasilescu, B., Serebrenik, A., and van den Brand, M. G. J. (2012). Who's who in Gnome: Using LSA to merge software repository identities. In *28th International Conference on Software Maintenance (ICSM)*, pages 592--595.
- Lavallée, M. and Robillard, P. N. (2015). Why good developers write bad code: an observational case study of the impacts of organizational factors on software quality. In *37th International conference on Software engineering (ICSE)*, pages 677--687.
- Lee, A., Carver, J. C., and Bosu, A. (2017). Understanding the Impressions, Motivations, and Barriers of One Time Code Contributors to FLOSS Projects: A Survey. In *39th International Conference on Software Engineering (ICSE)*, pages 187--197.

- Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. In *4th International Symposium on Software Metrics*, pages 20--32.
- Lin, B., Robles, G., and Serebrenik, A. (2017). Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *12th International Conference on Global Software Engineering (ICGSE)*, pages 66--75.
- López-Fernández, L., Robles, G., Gonzalez-Barahona, J. M., and Herraiz, I. (2006). Applying social network analysis techniques to community-driven libre software projects. *International Journal of Information Technology and Web Engineering*, 1:27--48.
- Love, R. (2010). *Linux kernel development*. Addison-Wesley, Boston.
- Ma, D., Schuler, D., Zimmermann, T., and Sillito, J. (2009). Expert recommendation with usage expertise. In *25th International Conference on Software Maintenance (ICSM)*, pages 535--538.
- Madey, G., Freeh, V., and Tynan, R. (2002). The open source software development phenomenon: An analysis based on social network theory. In *Americas Conference on Information Systems (AMCIS)*, pages 1806--1813.
- McDonald, D. W. and Ackerman, M. S. (2000). Expertise recommender: a flexible recommendation system and architecture. In *Conference on Computer Supported Cooperative Work (CSCW)*, pages 231--240.
- Meneely, A. and Williams, L. (2009). Secure open source collaboration: An empirical study of linus' law. In *16th Conference on Computer and Communications Security (CCS)*, pages 453--462.
- Meneely, A. and Williams, L. (2011). Socio-technical developer networks: Should we trust our measurements? In *33rd International Conference on Software Engineering (ICSE)*, pages 281--290.
- Meneely, A., Williams, L., Snipes, W., and Osborne, J. (2008). Predicting failures with developer networks and social network analysis. In *16th International Symposium on Foundations of Software Engineering (FSE)*, pages 13--23.
- Minto, S. and Murphy, G. C. (2007). Recommending emergent teams. In *4th Workshop on Mining Software Repositories (MSR)*, pages 5--5.

- Mistrík, I., Grundy, J., van der Hoek, A., and Whitehead, J. (2010). *Collaborative software engineering: challenges and prospects*, pages 389–403. Springer.
- Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346.
- Mockus, A. and Herbsleb, J. (2001). Challenges of global software development. In *7th International Software Metrics Symposium (Metrics)*, pages 182–184.
- Mockus, A. and Herbsleb, J. D. (2002a). Expertise browser: a quantitative approach to identifying expertise. In *24th International Conference on Software Engineering (ICSE)*, pages 503–512.
- Mockus, A. and Herbsleb, J. D. (2002b). Expertise browser: A Quantitative Approach to Identifying Expertise. In *24th International Conference on Software Engineering - (ICSE)*, pages 503–512.
- Nagappan, N., Murphy, B., and Basili, V. (2008). The influence of organizational structure on software quality: An empirical case study. In *30th International Conference on Software Engineering (ICSE)*, pages 521–530.
- Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88.
- Newman, M. E. J. (2003). Mixing patterns in networks. *Physical Review E*, 67:026126.
- Newman, M. E. J. (2004). Coauthorship networks and patterns of scientific collaboration. *Proceedings of the National Academy of Sciences*, 101:5200–5205.
- Padhye, R., Mani, S., and Sinha, V. S. (2014). A study of external community contribution to open-source projects on GitHub. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 332–335.
- Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2015). Mining version histories for detecting code smells. *Transactions on Software Engineering*, 41(5):462–489.
- Palomba, F., Tamburri, D. A., Serebrenik, A., Zaidman, A., Arcelli Fontana, F., and Oliveto, R. (2018). Poster: How do community smells influence code smells. In *40th International Conference on Software Engineering (ICSE)*.

- Panichella, S., Canfora, G., Penta, D. M., and Oliveto, R. (2014). How the evolution of emerging collaborations relates to code changes: an empirical study. In *22nd International Conference on Program Comprehension (ICPC)*, pages 177--188.
- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053--1058.
- Parnas, D. L. (1994). Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279--287.
- Passos, L., Padilla, J., Berger, T., Apel, S., Czarnecki, K., and Valente, M. T. (2015). Feature scattering in the large: a longitudinal study of Linux kernel device drivers. In *14th International Conference on Modularity*, pages 81--92.
- Pilato, C. M., Collins-Sussman, B., and Brian, W. F. (2008). *Version Control With Subversion*. O'Reilly, 2nd edition.
- Pinto, G., Steinmacher, I., and Gerosa, M. A. (2016). More Common Than You Think: An In-depth Study of Casual Contributors. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 112--123.
- Pinzger, M., Nagappan, N., and Murphy, B. (2008). Can developer-module networks predict failures? In *16th International Symposium on Foundations of Software Engineering (FSE)*, pages 2--12.
- Prikladnicki, R., Audy, J. L. N., and Shull, F. (2010). Patterns in effective distributed software development. *IEEE Software*, 27:12--15.
- Rahman, F. and Devanbu, P. (2011). Ownership, experience and defects. In *33rd International Conference on Software Engineering (ICSE)*, pages 491--500.
- Ray, B., Posnett, D., Filkov, V., and Devanbu, P. (2014). A large scale study of programming languages and code quality in GitHub. In *22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 155--165.
- Ricca, F. and Marchetto, A. (2010). Are heroes common in FLOSS projects? In *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1--4.
- Ricca, F., Marchetto, A., and Torchiano, M. (2011). On the difficulty of computing the truck factor. In *Product-Focused Software Process Improvement*, volume 6759, pages 337--351. Springer.

- Rigby, P. C., Zhu, Y. C., Donadelli, S. M., and Mockus, A. (2016). Quantifying and mitigating turnover-induced knowledge loss. In *38th International Conference on Software Engineering (ICSE)*, pages 1006--1016.
- Rodriguez-Bustos, C. and Aponte, J. (2012). How distributed version control systems impact open source software projects. In *9th Working Conference on Mining Software Repositories (MSR)*, pages 36--39.
- Schuler, D. and Zimmermann, T. (2008). Mining usage expertise from version archives. In *5th International Working Conference on Mining Software Repositories (MSR)*, pages 121--124.
- Shull, F., Singer, J., and Sjøberg, D. I. (2007). *Guide to Advanced Empirical Software Engineering*. Springer.
- Steinmacher, I., Conte, T., Gerosa, M. A., and Redmiles, D. (2015). Social Barriers Faced by Newcomers Placing Their First Contribution in Open Source Software Projects. In *18th Conference on Computer Supported Cooperative Work & Social Computing (CSCW)*, pages 1379--1392.
- Steinmacher, I., Conte, T. U., Treude, C., and Gerosa, M. A. (2016). Overcoming open source project entry barriers with a portal for newcomers. In *38th International Conference on Software Engineering (ICSE)*, pages 273--284.
- Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering (FSE)*, pages 99--108.
- Thongtanunam, P., McIntosh, S., Hassan, A. E., and Iida, H. (2016). Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *38th International Conference on Software Engineering (ICSE)*, pages 1039--1050.
- Torchiano, M., Ricca, F., and Marchetto, A. (2011). Is my project's truck factor low? In *2nd international workshop on Emerging trends in software metrics (WETSoM)*, pages 12--18.
- Tsay, J., Dabbish, L., and Herbsleb, J. (2014). Let's talk about it: evaluating contributions through discussion in GitHub. In *22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 144--154.

- Vasilescu, B. (2014). Human aspects, gamification, and social media in collaborative software engineering. In *36th International Conference on Software Engineering (ICSE)*, pages 646--649.
- Vasilescu, B., Filkov, V., and Serebrenik, A. (2015). Perceptions of diversity on GitHub: A user survey. In *8th International Workshop on Cooperative and Human Aspects of Software Engineerin (CHASE)*, pages 50--56.
- Vasilescu, B., Schuylenburg, S. V., Wulms, J., Serebrenik, A., and Brand, M. G. J. V. D. (2014a). Continuous integration in a social-coding world: empirical evidence from GitHub. In *30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 401--405.
- Vasilescu, B., Serebrenik, A., Goeminne, M., and Mens, T. (2014b). On the variation and specialisation of workload – a case study of the Gnome ecosystem community. *Empirical Software Engineering*, 19(4):955–1008.
- Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of small-world networks. *Nature*, 393:440--2.
- Wiese, I. S., da Silva, J. T., Steinmacher, I., Treude, C., and Gerosa, M. A. (2016). Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant. In *32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 345--355.
- Wilcoxon, F. (1992). *Individual Comparisons by Ranking Methods*, pages 196--202. Springer, New York, NY.
- Williams, L. and Kessler, R. (2003). *Pair Programming Illuminated*. Addison Wesley.
- Xu, J., Gao, Y., Christley, S., and Madey, G. (2005). A topological analysis of the open source software development community. In *38th Annual Hawaii International Conference on System Sciences*, pages 198a–198a.
- Yamashita, K., McIntosh, S., Kamei, Y., Hassan, A. E., and Ubayashi, N. (2015). Revisiting the applicability of the pareto principle to core development teams in open source software projects. In *14th International Workshop on Principles of Software Evolution (IWPSE 2015)*, pages 46--55.
- Yang, X. (2014). Social network analysis in open source software peer review. In *22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 820-822.

- Zazworka, N., Stapel, K., Knauss, E., Shull, F., Basili, V. R., and Schneider, K. (2010). Are developers complying with the process: an xp study. In *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 14:1--14:10.
- Zhang, W., Yang, Y., and Wang, Q. (2011). Network analysis of OSS evolution: an empirical study on ArgoUML project. In *12th International Workshop on Principles of Software Evolution (IWPSE)*, pages 71--80.
- Zhou, M. and Mockus, A. (2010). Developer fluency: Achieving true mastery in software projects. In *8th International Symposium on Foundations of Software Engineering (FSE)*, pages 137--146.