Empir Software Eng (2016) 21:642–683
DOI 10.1007/s10664-015-9406-4

From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open Source Software projects

Ingo Scholtes[1] · **Pavlin Mavrodiev**[1] · **Frank Schweitzer**[1]

Published online: 19 December 2015
© Springer Science+Business Media New York 2015

Abstract Complex software development projects rely on the contribution of teams of developers, who are required to collaborate and coordinate their efforts. The productivity of such development teams, i.e., how their size is related to the produced output, is an important consideration for project and schedule management as well as for cost estimation. The majority of studies in empirical software engineering suggest that - due to coordination overhead - teams of collaborating developers become less productive as they grow in size. This phenomenon is commonly paraphrased as *Brooks' law of software project management*, which states that "adding manpower to a software project makes it later". Outside software engineering, the non-additive scaling of productivity in teams is often referred to as the *Ringelmann effect*, which is studied extensively in social psychology and organizational theory. Conversely, a recent study suggested that in Open Source Software (OSS) projects, the productivity of developers *increases* as the team grows in size. Attributing it to collective synergetic effects, this surprising finding was linked to the Aristotelian quote that "the whole is more than the sum of its parts". Using a data set of 58 OSS projects with more than 580,000 commits contributed by more than 30,000 developers, in this article we provide a large-scale analysis of the relation between size and productivity of software development teams. Our findings confirm the negative relation between team size and productivity previously suggested by empirical software engineering research, thus providing quantitative evidence for the presence of a strong Ringelmann effect. Using fine-grained data on

Communicated by: Burak Turhan

✉ Ingo Scholtes
ischoltes@ethz.ch

Pavlin Mavrodiev
pmavrodiev@ethz.ch

Frank Schweitzer
fschweitzer@ethz.ch

[1] ETH Zürich, Chair of Systems Design, Weinbergstrasse 56/58, 8092 Zurich, Switzerland

the association between developers and source code files, we investigate possible explanations for the observed relations between team size and productivity. In particular, we take a network perspective on developer-code associations in software development teams and show that the magnitude of the decrease in productivity is likely to be related to the growth dynamics of co-editing networks which can be interpreted as a first-order approximation of coordination requirements.

Keywords Software engineering · Repository mining · Productivity factors · Social aspects of software engineering · Open source software · Coordination

1 Introduction

Most of today's software projects are so complex that they cannot be developed by a single developer. Instead, potentially large teams of software developers need to collaborate. This necessity of collaboration in large teams raises a simple, yet important question: How productive is a team of developers compared to a single developer? Or, in other words: How much time do n developers need to finish a software project compared to the time taken by a single developer? This question is of significant importance not only for project management but also for the development of reasonable cost estimation models for software engineering processes. One may naively assume that the productivity of individual team members is *additive*, i.e., that, compared to the time taken by a single developer, a team of n developers will speed up the development time by a factor of n. However, this naive perspective misses out two important factors that can give rise to a non-additive scaling of productivity.

First, the collaboration of developers in a team can give rise to *synergy effects*, which result in the team being *more productive* than one would expect from simply adding up the individual productivities of its members. Under this assumption, the average output per team member can be *increased* by simply adding developers to the team, a fact that has recently been related to Aristotle's quote that *"the whole is more than the sum of its parts"* (Sornette et al. 2014). A second, contrary factor that influences the productivity of software development teams is the communication and coordination overhead which is becoming more pronounced as teams grow larger. This increasing overhead can impact productivity and it has thus been discussed extensively in the software engineering and project management literature. In particular, it has been argued that coordination issues can lead to situations where the average output per team member *decreases* as the size of the team is increased. Studies showing that growing team sizes negatively affect productivity can be traced back to early studies of Maximilian Ringelmann, and the effect has accordingly be named the "Ringelmann effect" (Ringelmann 1913). In the context of software engineering, it can be related to *Brooks' law* of software project management, which states that *"adding manpower to a late software project makes it later"*, and Brooks rejects the idea of additive, or even super-additive productivity by adding that "nine women can't make a baby in one month" (Brooks 1975).

Apart from very few exceptions, there is a broad consensus in the software engineering literature that the size of software development teams, both in traditional closed source and open source environments negatively affects the average productivity. However, quantitative evidence for this fact that would go beyond small-scale case studies or surveys is relatively sparse. At the same time, there is significant confusion over the question which quantitative

indicators can reasonably be used to measure the productivity of software development teams or individual developers.

Using a large-scale data set covering the full history of 58 Open Source Software (OSS) projects hosted on the social coding platform GITHUB, in this paper we quantitatively address the question of how the size of a software development team is related to productivity. Based on a time-slice analysis of more than 580,000 commit events over a period of more than 14 years, we analyse the output of projects in terms of code and study how their time-varying productivity relates to the number of active software developers. The contributions of our empirical analysis are as follows:

1. Using the *distribution of inter-commit times*, we first identify reasonable time windows for the definition of team size and the analysis of commit activities in OSS projects.
2. Based on a microscopic, textual analysis of commit contents, we measure the contributions of individual commits and highlight their huge variance. By this, we quantitatively prove that the mere *number of commits* should not be used as a reasonable measure of productivity, and that an analysis of actual commit content is needed.
3. We define a more reasonable measure for the contribution of developers which is based on the so-called *Levenshtein edit distance* (Levenshtein 1966) between consecutive versions of source code files.
4. Using this fine-grained measure of code contributions, we quantitatively show that in all of the studied OSS projects the average productivity of developers decreases as the team size increases, thus providing quantitative evidence for the Ringelmann effect.
5. Addressing possible mechanisms behind the Ringelmann effect, we finally take a network perspective on evolving coordination structures that is based on a fine-grained analysis of *diffs* between commits occurring within certain time windows. In particular, using the association between source code regions and developers, we construct time-evolving co-editing networks. These networks can be seen as language-independent first-order approximations for coordination structures that can be constructed solely based on widely available repository data.
6. Finally, we analyse the growth dynamics of co-editing networks constructed from repository data. For all projects in our data set, we observe a *super-linear* growth of co-editing networks, which can be seen as one potential mechanism behind the observed Ringelmann effect.

Using a large and open data set, our study validates the common assumption in software engineering that the size of a team *negatively* affects the average productivity of its members. This highlights the fact that, possibly due to the duplication of efforts, increasing coordination overhead, as well as decreasing accountability, software development projects represent *diseconomies of scale*.

We argue that both our results as well as our methodology are useful to refine and calibrate existing software development cost models based on empirical data from software development repositories. Our investigation of commit numbers and the size distribution of commits further calls for a cautious use of commit-based productivity measures, since their naive application can easily yield erroneous results.

The remainder of this paper is structured as follows. In the following Section 2 we carefully review the extensive body of literature on team productivity which has been published in software engineering, but also in fields such as *sociology*, *social psychology* and *organizational theory*. We then continue by providing a detailed description of our data set, our time-slice analysis and our measure for productivity in Section 3. Empirical results on

the scaling of productivity are presented and discussed in Section 4. Investigating potential mechanisms behind the observed scaling relations, in Section 5 we take a network perspective on the association between developers and edited source code regions in OSS projects. Having discussed potential threats to validity and future work in Section 6, we conclude our paper in Section 7.

2 Background and Related Work

We start our investigation by a review of the existing body of literature on productivity factors in software development projects. Rather than limiting our focus on the field of empirical software engineering, we additionally review works studying how the size of teams, groups or organizations affects their performance which have been completed in fields such as *sociology*, *social psychology*, *management science* or *organizational theory*.

In what has since been named one of the foundational experiments of social psychology, the French agricultural engineer Maximilian Ringelmann (Kravitz and Martin 1986; Ringelmann 1913) quantitatively studied the efficiency of a group of human workers jointly pulling on a rope. He showed that the individual performance decreased with increasing group size, an effect later termed as "Ringelmann effect" in social psychology (Ingham et al. 1974). Ringelmann attributed the effect mainly to the increasingly challenging coordination in larger groups. He additionally highlighted the potential influence of social factors that may affect the motivation of individual group members (Steiner 1972). These social factors have since been investigated in detail in a number of studies from psychology, organizational theory and management science. Here, the tendency of group members to spend less effort in larger groups has been named "free-riding", the "sucker effect" or "social loafing" (Shepperd 1993). It is generally attributed to situations characterized by a lack of individual accountability and shared responsibility (Latane et al. 1979; Williams et al. 1981; Williams and Karau 1991; Wagner 1995) and it has been shown to be a robust phenomenon in various contexts and - with varying strength - across cultures (Yetton and Bottger 1983; Jackson and Harkins 1985; Earley 1989; Karau and Williams 1993, 1995; Chidambaram and Tung 2005; Shiue et al. 2010.)

Notably, both the coordination challenges and the motivational factors which have been investigated as potential mechanisms behind the Ringelmann effect, play an important role in software development. Especially in large and fluid teams with no fixed assignment of responsibilities, based on findings from social psychology, the lack of accountability could potentially negatively affect the amount of contributions. Furthermore, complex collaborative tasks such as software development entail a need for coordination which is likely to become more challenging as the size of the team increases.

If and how these factors affect the productivity of software development teams has been studied extensively since the early days of empirical software engineering research. According to *Brooks' law* of software project management (Brooks 1975), the size of a development team negatively affects its productivity. Brooks' rationale behind this "law" is based on two main factors: (i) the significant "ramp-up" time needed by new developers before they become productive, and (ii) the super-linear coordination cost which is due to the quadratic scaling of the number of possible communication channels.

These intuitive arguments have been substantiated by a a number of empirical studies showing that the size of the development team negatively affects software development

productivity. Based on a survey with 77 software developers from eleven software companies, Paiva et al. (2010) argue that developers assess the size of a development team as one of those factors that have the strongest negative impact on productivity. Blackburn et al. (1996) empirically studied software development projects in Japan, the United States and Western Europe across a time period of four years. The authors found a negative correlation between the productivity of software development teams and their size, referring to the phenomenon as the "productivity paradox". Using data on 99 software projects of the European Space Agency with a total of more than 5 Million lines of code, Maxwell et al. (1996) studied a number of factors influencing productivity. The authors found that productivity significantly decreases with increasing team size and argue that this "is probably due to the coordination and communication problems that occur as more people work on a project." (Maxwell et al. 1996).

A first step to investigate the coordination cost associated with larger development teams was taken by works studying how work is distributed among team members. Using the Open Source projects APACHE HTTPD and MOZILLA FIREFOX as case studies, Mockus et al. (2000) and (2002) found that a small core of developers is responsible for the vast majority of code changes. They further argue that in OSS projects "communication and coordination overhead issues [...] typically limit the size of effective teams to 10-15 people" (Mockus et al. 2000). The finding that the majority of code is developed by a small core group of developers has been validated in a number of empirical studies studying the distribution of code distributions (Lerner and Tirole 2002; German 2006).

A number of further works have studied the question of how coordination costs affect software development productivity. Analyzing the congruence between coordination patterns and coordination needs in software development teams, the fact that a lack of coordination negatively affects software development productivity was shown in (Cataldo et al. 2008; Cataldo and Herbsleb 2013). The scaling of coordination costs and productivity in OSS projects was studied by Adams et al. (2009), finding a non-linear scaling that is in line with earlier studies suggesting an optimum team size. In particular, the authors identify a first regime with comparable coordination effort for development teams consisting of less than ten developers. A second regime in which the growing number of developers does not lead to a proportionate increase of coordination efforts is likely due to the emergence of modular team structures which mitigate coordination efforts. A third regime is identified beyond a critical team size, in which coordination efforts quickly increase in a super-linear fashion. The negative impact of coordination requirement observed in empirical studies has finally been integrated in a number of *cost estimation models*, such as the *Constructive Cost Model* (COCOMO) or its successor COCOMO II (Boehm 1984; Boehm et al 2000), which are commonly used in project management.

Apart from social psychology and empirical software engineering, *management science* and *organizational theory* are two further fields of study which have extensively addressed the question of how the productivity in a project is related to its size. Here, the question how the productivity, or cost, of software projects is related to their size is commonly rephrased in terms of whether software projects are *economies* or *diseconomies of scale* (Stigler 1958). This question has been addressed from different perspectives.

Measuring project size in terms of the size of the code base and productivity in terms of person hours, Premraj et al. (2005) studied more than 600 software projects from Finnish companies, finding no significant evidence for a non-linear relation between size of the code base and productivity. A series of other works has argued that there exist both economies and diseconomies of scale in the context of software development (Banker and Kemerer 1989; Banker et al. 1994; Banker and Slaughter 1997; Banker and Kauffman 2004).

Notably, evidence for *economies* of scale has predominantly been found in software maintenance projects, which has been related to the possibility of batching modification requests (Banker and Slaughter 1997). In a large-scale study comprising more than 4,000 industrial software development projects, Comstock et al. (2011) studied the relation between team size and the effort required to complete a project. Consistent with *Brooks' law*, they find that the time taken to develop a software does not decrease proportionally with team size, concluding that "doubling the team size does not halve the time required" (Comstock et al. 2011). In line with earlier works showing a non-linear and non-monotonous relationship between team size and coordination costs (Adams et al. 2009), they further argue that there exists an optimum team size that depends on project characteristics like project size as well as other productivity parameters.

Due to the availability of fine-grained data containing project variables like cost, effort, personnel and project performance, as well as the need for cost estimation models, the majority of studies on software productivity, including the large-scale study performed by Comstock et al. (2011), have studied industrial, and thus mainly closed-source software projects. Fewer studies in management science have specifically addressed Open Source Software communities, addressing the open question whether OSS projects are economies or diseconomies of scale (von Krogh et al. 2003). Taking an economic perspective and defining productivity as the real value contributed per employee, Harison and Koski (2008) compared the productivity of Open Source Software firms with those of companies developing proprietary solutions. While they did not study productivity in terms of code production, the authors find employees in OSS-based companies to be less productive from an economic point of view. Disagreeing with studies showing diminishing returns to scale prevalent in the software engineering literature, Sornette et al. (2014) argued that the productivity of OSS communities increases with team size in a *super-linear fashion*. Interpreting "commits" as "commitment", the authors define the productivity of a project as the total number of commits made within five day time windows and find that a large number of contributors facilitates bursts of commit activity.

Summarising the large body of interdisciplinary works outlined above, we conclude that there is broad consensus in the software engineering community that increasing the size of software development teams - at least beyond certain, comparably small numbers - *negatively* affects productivity. At the same time, there is an ongoing debate of whether *Open Source Software projects* are economies or diseconomies of scale, results critically depending on (i) the type of software projects considered (i.e., software development vs. maintenance projects), and (ii) the operationalization of project size and productivity. Regarding the second aspect, it is important to note that recent results on a super-linear scaling of productivity in OSS communities have been obtained using a productivity measure that is based on the total number of commits, thus neglecting the actual commit contribution in terms of code.

Contributing to this debate, in our work we first quantitatively show that this approach of measuring productivity introduces a systematic error which results in an overestimation of productivity that is due to (i) the skewed commit size distribution, and (ii) basic software engineering and collaboration practices such as "commit early commit often" which are likely to result in commit sizes becoming smaller as teams grow in size. Defining a measure of productivity that encompasses both commits and - based on a fine-grained textual analysis - the actual *contents* of these commits, we show that Open Source Software Communities represent diseconomies of scale, in which the average productivity of developers decreases as the team size grows. With this work, we contribute

to the ongoing discussion on the scaling of productivity in software development projects, showing that earlier results on the presence of the Ringelmann effect in industrial software projects also extend to Open Source Communities. Our study further highlights the importance for complex systems and data science studies to take into account basic principles of collaborative software engineering, thus contributing to a knowledge exchange between the disciplines of complex systems, management science, and empirical software engineering.

3 Methodology

We study the research questions outlined above using a large-scale data set collected from the collaborative software development platform GITHUB. It offers a free, web-based hosting service for Open Source Software repositories centered around the distributed version control system GIT. With more than 5 Million developers collaborating on more than 10 Million repositories, GITHUB has recently become the most popular software project hosting platform in the world (Gousios et al. 2014). This popularity is partly due to the fact that, in addition to a mere version control of source code, GITHUB offers a number of *social features* such as issue tracking, a wiki system and team management which support collaboration and coordination. A feature that makes GITHUB particularly popular in the Open Source community is the ability to easily *fork* the repositories of others. In a nutshell, this allows users to copy the source code of existing projects into their own repository, instantly enabling them to modify and advance the code base without the need to coordinate the changes with the original developers. Changes that may be useful for others can then be propagated back to the original repository via a so-called *pull request*, i.e., a request sent to the maintainers of the root repository to pull the changes from the forked repository and *merge* them into the main code base.

3.1 Data Set

For our study, we collected the full commit history of 58 OSS projects from the publicly available GITHUB API.[1] The initial choice of projects was based on the following criteria: (i) the project should be among the 100 most frequently forked projects, (ii) it should still be active, showing commit activity in the week when the data were collected, (iii) there should be at least 50 different active developers across the whole project history, and (iv) there should be at least one year of activity. The data on these 58 projects amount to a total of 581,353 commits contributed by 30,845 different users, with an average of 10,023 commits and 531 unique developers per project. A detailed summary of our choice of projects, the programming language used, the time period covered, as well as the number of commits and developers is shown in Table 3 in the supplementary information (SI).

In the GIT version control system the development history of a project consists of a tree of commits, with the first commit representing the root and each subsequent commit storing a *parent* pointer to the commit immediately preceding it. In this way, starting

[1] see https://developer.github.com/v3/

from the most recent commit of a project, the whole development history can be traced back by following the parent pointers. The GITHUB API returns information about each commit and the corresponding GIT tree. More specifically, for each commit our data set includes:

- an SHA hash uniquely identifying the commit
- the name and email of the developer who authored the commit
- the time stamp of the commit (with a resolution of seconds)
- the list of files changed by this commit
- the *diffs* of all files changed by the commit, which allows to reconstruct the precise changes to the source code at the level of individual characters
- the parent pointer (SHA hash) of the preceding commit

When project maintainers merge pull requests (i.e., code authored by others) from forked repositories, a special *merge* commit can be created in the commit tree. We exclude such commits from the analysis as they would wrongly attribute the diffs to the maintainer who executed the merge and not to the developer who authored the pull request. For technical details, we refer the reader to Section A2 in the Appendix.

We make both the raw and the preprocessed data used in our study freely available (Scholtes et al. 2015). In the following we provide a detailed description of how we use this data set to analyse the relation between the size of a software development team and its productivity.

3.2 Measuring Team Size

Understanding the relation between team size and productivity first requires a reasonable definition for the size of the development team of an OSS project. Being informal and fluid social organisations, the simple question *who* belongs to the development team of an Open Source project at a given point in time is not trivial to answer since there is - in general - no "formalized" notion of who is a member of the project. This loose notion of team membership is particularly pronounced in GITHUB, since the integration of *pull requests* allows users to contribute code without the need for a prior procedure of obtaining commit rights in the repository.

Using time-stamped commit data, a first naive approach to study team size could be based on the analysis of *activity spans* of developers, i.e., taking the first and last commit of each developer to the project and considering them as team members in between the time stamps of those two commits. However, this simple approach generates several problems: First, developers may leave the project, be inactive for an extended period of time and then rejoin the project later. Secondly, our analysis is necessarily based on a finite observation period. Close to the end of this observation period, the naive approach outlined above will wrongly consider that developers left the project after their last commit, even though in reality many of them are likely to commit again in the future. Finally, Open Source Software projects feature a large number of *one-time contributors* who - using this simple approach - will not be considered as project members even though they both contribute to the development and impose coordination costs. Figure 14 in the SI shows that this is indeed the case for our data set, where more than 40 % of all commits in the majority of the projects have been contributed by one-time contributors.

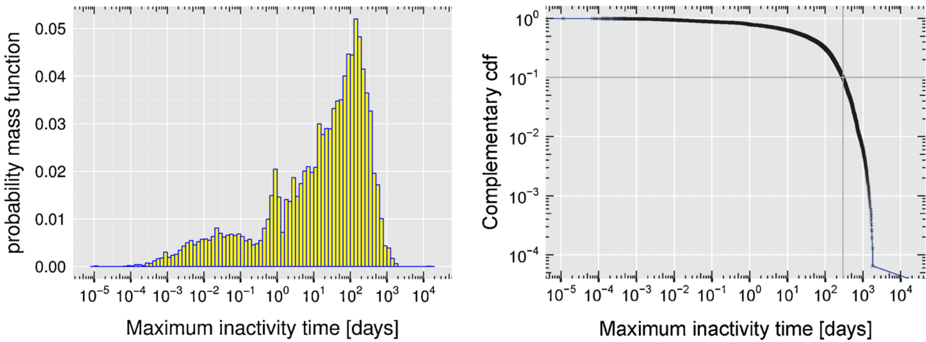

Fig. 1 Histogram and complementary cumulative distribution function of maximum inactivity times aggregated across all projects

We avoid these problems by taking into account the intermittent nature of developer activities. In particular, for each individual developer we analyse his/her whole commit history and calculate the maximum time of inactivity between any two consecutive commits.[2] Doing this for all developers yields a *maximum inactivity time distribution*, which allows us to estimate the probability with which a developer who has been inactive for a certain time will commit again in the future. The left panel of Fig. 1[3] shows the histogram of maximum inactivity times of all developers, aggregated across all projects. The right panel shows the complementary cumulative distribution function (ccdf) of the distribution. Here, we observe that 90 % of all consecutive commits occur within time periods of less than 295 days. In other words, the chance of a developer committing again after having been inactive for more than 295 days is less than 10 %.

Based on this finding, we utilize a a sliding window with a size of 295 days to define the size of the development team at any given time t.[4] Precisely, at time t we define the team T_t of a project to consist of all developers who have committed at least once within the time window $[t - \delta_{\text{team}}, t]$ of window size δ_{team}.

3.3 Measuring Software Development Productivity

Apart from a reasonable definition of the development team of a project, a second major building block of our study is the definition of software development productivity. In general, the productivity of an individual or an organization is defined as the output produced per time unit. In our study we exclusively focus on the output in terms of source code artifacts produced by software developers. A number of earlier studies have used similar approaches to study the productivity of software development teams based on data from software repositories: The simplest possible approach to measure productivity

[2]Notably, we avoid a bias towards large inactivity times, by not taking into account developers who committed only once.

[3]please note the log scale on the x-axis.

[4]effectively, we accept a less than 10 % chance of falsely excluding from the team at time t a developer who eventually commits after more than 295 days of inactivity.

is to calculate the total number of commits in a given time period (Adams et al. 2009; Sornette et al. 2014). However, this approach introduces multiple problems: First and foremost, a number of prior studies have shown that the size of commits follows a highly skewed distribution (Robles et al. 2004; Hindle et al. 2008; Alali et al. 2008; Arafat and Riehle 2009; Hofmann and Riehle 2009), arguing that it is typically a very small number of large commits which are fundamentally important in the evolution of software, while the majority of commits are minor contributions. As such, the *size of a commit* has to be taken into account when measuring the productivity of developers in software projects.

Even more important for the purpose of our study, using the *total number of commits* introduces a *systematic error* when studying the relation between the number of developers in a development team and the total number of commits: First, the total number of commits contributed by n developers active in a given time period cannot - by definition - be less than n, which is why the total number of commits must scale *at least* linearly with team size. Secondly, due to basic principles guiding the use of revision control systems, developers in larger development teams tend to avoid generating conflicts by splitting their contributions in increasingly atomic changes, a strategy commonly paraphrased as “commit early, commit often”.

From the arguments above, we conclude that the mere *number of commits* of a developer is not a good proxy for the developer’s contribution, unless the *size of those commits* is taken into account. In our study we thus leverage the fact that our data set contains not only the occurrence of commits, but also the associated *diff* records which allow us to analyse the *contents* of commits.

Several methods can be used to turn this information into a quantitative measure of productivity. A simple approach is to use *diff* records in order to calculate the number of lines changed in a file, then calculating the commit size as the total number of lines changed in all affected files (Gousios et al. 2008). This methods is a significant improvement over using the mere number of commits. However, it still suffers from the fact that not all line changes are equal. In particular, following this approach, commenting out 100 lines of source code is seen as the same contribution as writing those 100 lines in the first place. For this reason, other standard measures of software development productivity take into account the semantics by extracting so-called *function points* (Albrecht 1979), i.e., larger units of source code that are directly related to some functionality of the software. Since the source code needs to parsed in order to extract these function points, this approach necessarily depends on the programming language and does not allow for an easy comparison between projects developed in different languages.

Our data set contains projects written in a mix of different programming languages (see Table 3 for details), which requires us to quantify productivity in a language-independent way. Rather than counting commits or the changed lines of code, we take a fine-grained approach which captures the changes of the source code at the level of individual characters. Precisely, for each commit we extract the diff records of all affected files, which allows us to unambiguously reconstruct the source code before and after the commit. We then calculate the *Levenshtein edit distance* (Levenshtein 1966) between these versions of the source code. The Levenshtein distance captures the number of single-character deletions, insertions and substitutions required to change the version *before* the commit into the version *after* the commit. For each of the more than 580,000 commits in our data set, we then sum the Levenshtein edit distances of the files affected by a commit, defining this sum as the *commit contribution*.

Apart from taking into account the size, or contribution, of commits, another important aspect in the analysis of time-stamped commit data is that productivity evolves in time. As

such, we need to be able to define the productivity of a project within a particular time window. In order to define the productivity P_t at time t, we consider all commits occurring within a window $[t - \delta_{\text{prod}}, t]$ of window size δ_{prod}. On the one hand, the time window should be long enough to contain a sufficiently large number of commits. On the other hand, the time window should be short enough such that it allows us to get an idea about the instantaneous productivity at, or around, time t. Balancing these two requirements, we can again address the optimum choice of the time window by studying the activity patterns of developers.

Here, for each project we analyze the time line of *all* commits (i.e., from *any* developer) and we calculate the time difference between any two consecutive commits. Doing this for all commits in all projects, we obtain the *inter-commit time distribution*. This helps us to understand how likely it is that multiple commits fall within a time window of a particular size. The histogram of inter-commit times is shown in the left panel of Fig. 2. As evident from the figure, the time difference between two consecutive commits varies over several orders of magnitude, with pronounced peaks around time differences of 30 minutes (i.e., ≈ 0.02 days) as well as around time differences of one day. The right panel of Fig. 2 shows the complementary cumulative distribution function (ccdf) of the inter-commit time distribution. From this distribution, we can deduce that in 90 % of the cases the time difference between consecutive commits of a project is less than 6 days.

Furthermore, we expect a strong weekly periodicity in commit activity due to the effect of workdays and weekends. To ensure that this effect is equally pronounced in all time windows, we choose the window size as a multiple of 7 days, i.e., $\delta_{\text{prod}} = 7$.

With the so defined productivity time window, in Fig. 3 we display the relation between development productivity as measured simply by the number of commits and by the Levenshtein edit distance. We segment the commit history of each project in consecutive, non-overlapping productivity windows of 7 days. For each window, we then record the total number of commits and the mean commit contribution, computed as the sum of the commit contributions of all commits divided by the number of commits. We use the mean instead of the total commit contribution in order to eliminate the systematic linear scaling arising from the fact that n commits necessarily have a total contribution of at least n characters.

Figure 3 shows that the mean commit contribution varies across several orders of magnitude for any number of commits observed in our data set. In the small commit ranges

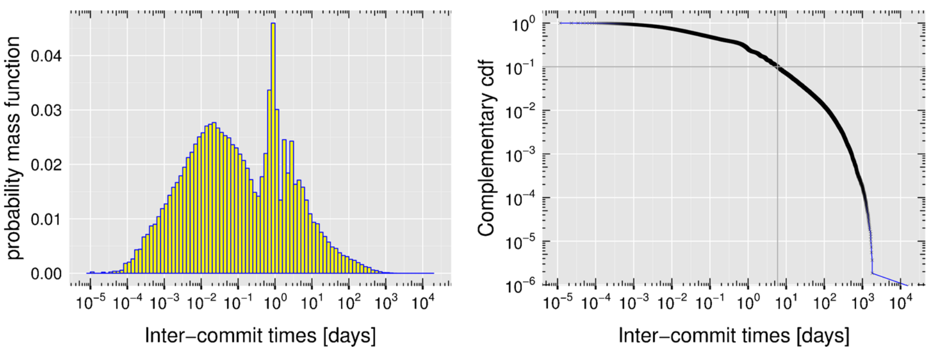

Fig. 2 Histogram and complementary cumulative distribution function of inter-commit times aggregated across all projects

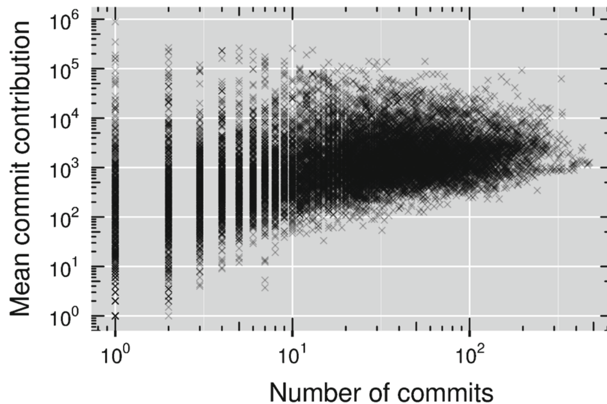

Fig. 3 Number of commits within a productivity time window versus the mean *commit contribution* of these commits, measured in terms of the Levenshtein edit distance. An alpha channel has been added to the plot for easier visualization of point densities

we observe variations of up to 6 orders of magnitude,[5] and even though the fluctuations decrease, the high commit range still exhibits variations of close to 3 orders of magnitude. This indicates that the Levenshtein edit distance cannot be replaced as a productivity measure by the total number of commits, without losing a significant amount of information on the contributions of those commits.[6]

3.4 Temporal Analysis of Productivity and Team Size

The procedure described above allows us to quantitatively assess the total production of source code, as well as the team size, i.e., the number of developers involved in the source code production at any given point in time. For each project, we use this methodology for a temporal analysis of the full history of time-stamped commit actions as follows:

Using the productivity time window of 7 days, we first segment the history of the project into consecutive, non-overlapping time slices of one week. For each productivity window reaching up to time t we calculate the total source code production based on the cumulative Levenshtein edit distance of all commit actions occurring within the interval $[t - 7d, t]$. We further compute the associated size of the developer team as the number of different developers who have committed at least once within the time interval $[t - 295d, t]$. We illustrate this procedure in Fig. 4.

Using this method, for each week in the history of all 58 projects in our data set, we obtain a pair of values capturing the *team size* and the *production* of commit contributions. This allows us to quantitatively analyse the relation between team size and productivity in the following section.

[5] which is commonly due to the initial stages of a project when a relatively large code base is submitted with the first few commits.

[6] Note that even though a regression model may produce a statistically significant relation between these two measures, due to the large variation, such a result should not be mistaken as evidence that the mean commit contribution can be replaced by the number of commits.

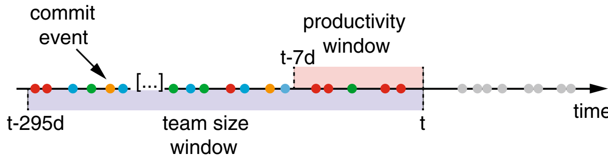

Fig. 4 Illustration of our methodology for the temporal analysis of productivity and team size: The total productivity is calculated as the sum of the Levenshtein edit distances of the five commits by two developers (*red and green*) in the productivity window. The team size of four is calculated as the number of different developers (*red, green, blue and orange*) who committed at least once within the team size window

4 Quantitative Analysis of Team Size and Productivity

In the following, we present the results of our quantitative analysis of developer productivity in OSS projects. Notably, we first study productivity by exclusively using the *productivity time window* of 7 days introduced above. For each project in our data set, we split the entire commit history into subsequent productivity time windows. For each time window we aggregate all commits by all developers active in that particular time window. For each productivity window, we additionally calculate the number of *active developers*, i.e., the number of those developers committing at least once within a productivity time window. Compared to the estimation of the team size introduced above, this initial analysis is thus based on a rather restrictive definition for the size of the development team. We include it here to rule out the possibility that our results sensitively depend on the choice of the team size window.

The left panel of Fig. 5 shows the relation between the *mean number of commits* per active developer and the number of developers active in the productivity time window, aggregated over all projects. For small numbers of active developers, the mean number of commits per developer exhibits a large variance over more than two orders of magnitude. As the number of active developers increases, one observes a decrease in the mean number of commits per developer. In particular, for projects with more than 50 developers committing in a given week, we observe a mean number of about two commits per week, while the mean number of commits for projects with a single active developer is about ten times as large. Even if one were to consider the mean number of commits as a proxy

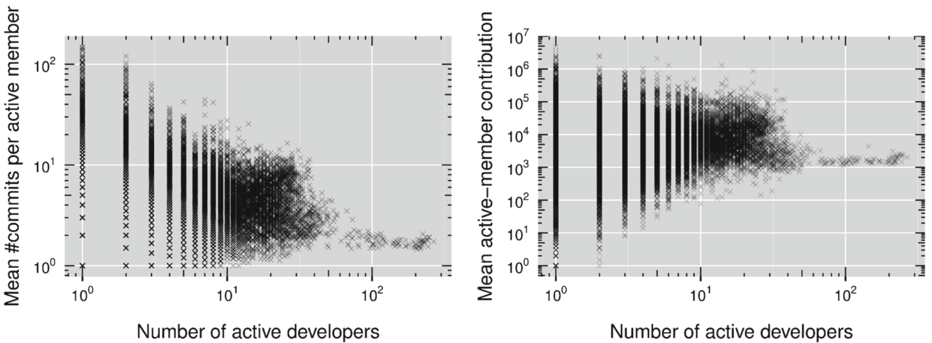

Fig. 5 Mean number of commits (*left*) and commit contribution (*right*) per active developer depending on the active developers, i.e., the number of developers committing at least once within a given *productivity time window*. An alpha channel has been added to the plot for easier visualization of point densities

for developer productivity, this finding is inconsistent with an economy of scale in collaborative software development, recently found by a similar analysis on a different data set (Sornette et al. 2014).

In Section 3 we have argued that different commits can have vastly different *commit contributions*, when measuring the contribution of a commit in terms of the actual code committed. In our analysis, we account for these differences by computing, for each commit, the *commit contribution* in terms of the Levenshtein edit distance between the versions of the source code before and after the commit. The total contribution of all developers active in a given productivity time window can then be computed as the cumulative Levenshtein edit distance of all commits occurring in that time window. The right panel of Fig. 5 shows the relation between the mean commit contribution per active developer and the number of active developers, i.e., the developers committing within the same productivity time window. We again aggregate over all 58 projects. Similar to the left panel of Fig. 5, a large variance can be observed for small numbers of active developers. Furthermore, we do not find evidence for a super-linear increase in total productivity, which would translate to an increase in the mean developer contribution. Similar to the left panel of Fig. 5, we rather observe that for projects with more than 50 active developers, the mean developer contribution is about 10^3 characters while it is - on average - at least one order of magnitude larger for projects with a small number of active developers.

In Section 3 we argued that the number of developers committing within a time window of 7 days does not provide us with a reasonable estimate for the size of the development team. This is due to the inactivity time distribution shown in Fig. 1, in which we observe a pronounced peak only around approximately 110 days. In particular, this shows that developers who have not been active in a given week should still be considered members of the development team, as they are still likely to commit again in the future. To mitigate this problem, we repeated our analysis using the window sizes illustrated in Fig. 4. For a given time t we again computed the mean number of commits, as well as the mean developer contribution with a productivity time window, i.e., within the time frame $[t - 7d, t]$. For the estimation of the *team size* we then count all developers who committed within the team size time window, i.e., in the time range $[t - 295d, t]$. In other words, we do not consider

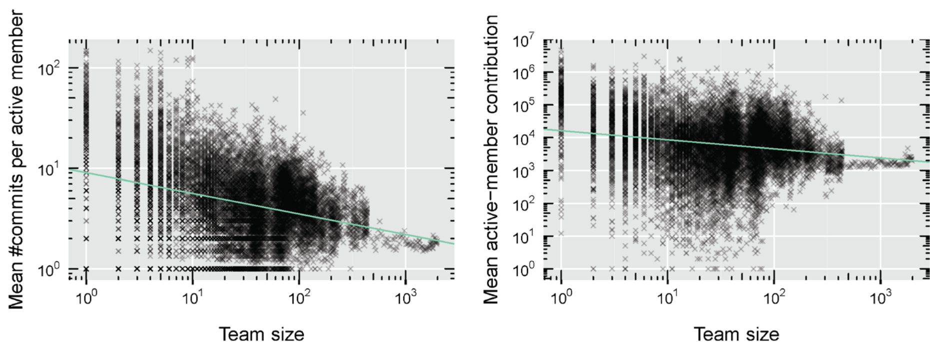

Fig. 6 Mean number of commits (*left*) and commit contribution (*right*) per active developer depending on the team size, i.e., the number of developers committing within a given *team size window*. *Green lines* indicate the fitted slopes α_0 and α_1 (see Section 4.1). An alpha channel has been added to the plot for easier visualization of point constellations

Table 1 Estimation of two linear models for Fig. 6

β_0	α_0	r^2	β_1	α_1	r^2
1.01±0.02	−0.24±0.01	0.16	4.35±0.04	−0.36±0.02	0.08

MM-estimation was used to estimate the coefficients of the regressors. The coefficients are presented together with their corresponding 95 % confidence intervals and are highly significant at $p < 0.001$. The sample size for both models is 13998

developers to be members of the development team if they have not committed for more than 295 days.

The results of this analysis are shown in Fig. 6. The left panel shows the relation between the mean number of commits per active developer and the team size. Similar to Fig. 5, we observe a decreasing mean number of commits as the teams size increases, which can be interpreted as evidence for the Ringelmann effect. Finally, the right panel of Fig. 6 shows the relation between the size of the development team and the mean commit contribution per active developer, again aggregated across all projects. Comparing small team sizes ($\approx 1-10$ developers) to large teams (> 300 developers), one observes a significant *decrease* in the mean developer contribution, on average dropping by more than two orders of magnitude. That the decrease in the mean commit contribution is even more pronounced than for the mean number of commits per developers indicates (i) that developers in large development teams tend to commit less often, and (ii) that these commits tend to be smaller in terms of the Levenshtein edit distance. Again, this finding can be interpreted as evidence for the Ringelmann effect or - in economic terms - for the fact that collaborative software engineering projects represent *diseconomies of scale*. It furthermore quantitatively substantiates common software engineering wisdom, as paraphrased for instance by *Brooks' Law* of software project management.

4.1 Scaling Factors of Productivity

So far, our arguments about the scaling of productivity with the size of the development team have been mostly visual. In the following, we substantiate these arguments by means of a regression analysis. For $\langle n \rangle$ being the mean number of commits per active developer, $\langle c \rangle$ being the mean commit contribution per active developer, and s being the team size as defined above, we can perform a regression analysis using the following two log-transformed linear models:[7]

$$\begin{aligned} \log\langle n \rangle &= \beta_0 + \alpha_0 \cdot \log s \\ \log\langle c \rangle &= \beta_1 + \alpha_1 \cdot \log s \end{aligned} \tag{1}$$

Table 1 shows the estimated coefficients of these models, inferred by means of a robust linear regression. The observed negative values for the scaling factors $\alpha_0 = -0.24$ and $\alpha_1 = -0.36$ quantitatively confirm the negative relation between team size and mean production of active developers previously observed visually in Fig. 6.

The low r^2 values reflect the high variability in the data, resulting from the inherent sensitivity of the Levenshtein edit distance. In essence, our regression models allow us to

[7] all logarithms are in base 10.

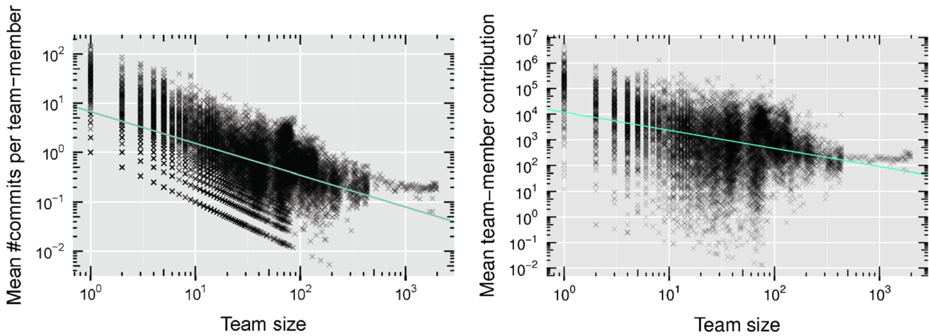

Fig. 7 Mean number of commits (*left*) and commit contribution (*right*) per team member, depending on the team size, i.e., the number of developers committing within a given *team size window*. *Green lines* indicate the fitted slopes α_2 and α_3. An alpha channel has been added to the plot for easier visualization of point constellations

infer a significant negative trend, but prevent us from making predictions about the mean production, given the team size[8] (c.f. further arguments in Section A5 in the Appendix).

Importantly, in all of our results so far, we computed the *mean* production (either in terms of the number of commits or commit contribution) per *active developer*, i.e., we divided the total production in a given productivity window by the number of developers who were committing at least once within this time window. When studying the relation between team size and mean production, one can alternatively compute a *mean production per team member* by dividing the total production by the team size, rather than the number of active developers. This is justified when interpreting the team size (i.e., all developers that have committed in the past team size window) as the amount of *resources* available to a project, and when considering productivity as the ratio between the generated outputs and these resources.

It furthermore emphasizes the ability of a project to continuously engage members of the development team, while a large number of inactive team members results in a decrease in observed productivity. In Fig. 7 we thus provide results for the mean production per team member. In both panels, we observe a significant decrease in terms of the mean number of commits (left) and the mean commit contribution (right) per team member, as the size of the team grows.

Using the data shown in Fig. 7, we can again perform a regression analysis. Like above, we use the following two log-transformed linear models

$$\begin{aligned} \log\langle n' \rangle &= \beta_2 + \alpha_2 \cdot \log s \\ \log\langle c' \rangle &= \beta_3 + \alpha_3 \cdot \log s \end{aligned} \tag{2}$$

where $\langle n' \rangle$ and $\langle c' \rangle$ denote the mean number of commits and the mean contribution *per team member* respectively. Table 2 shows the results of estimating the coefficients of this model by means of a robust linear regression.

Again we find negative values for the scaling factors α_2 and α_3 which indicate a negative relation between team size and mean productivity. Importantly, with this we confirm a negative scaling of productivity both for the *active developers*, which we computed using a

[8]Please note that we avoid regressing binned averages of $\log\langle c \rangle$ (or $\log\langle n \rangle$) as this would reduce the high variability in the data and would thus yield a spuriously large value of r^2.

Table 2 Estimation of two linear models for Fig. 7

β_2	α_2	R^2	β_3	α_3	R^2
0.95±0.03	−0.75±0.01	0.44	4.28±0.05	−0.86±0.02	0.25

MM-estimation was used to estimate the coefficients of the regressors. The coefficients are presented together with their corresponding 95 % confidence intervals and are highly significant at $p < 0.001$. The sample size for both models is 13998

window size of 7 days, as well as for the *team size*, which we computed for a window size of 295 days based on the maximum inactivity time statistics of our data set. This shows that the presence of a negative scaling does not sensitively depend on the window size used to calculate the number of contributors to a project.

Studying the *total* rather than the *mean* output produced, we can alternatively represent the negative relation between team size and mean productivity in terms of two simple Cobb-Douglas production functions that take the following form

$$\begin{aligned} N &= \beta_4 \cdot s^{\alpha_4} \\ C &= \beta_5 \cdot s^{\alpha_5}. \end{aligned} \tag{3}$$

Here N and C denote the *total* production, measured in terms of the total number of commits (N) and the total commit contributions (C) by all members of the development team. A log-transformation of these production functions yields the linear models

$$\begin{aligned} \log N &= \beta_4 + \alpha_4 \cdot \log s \\ \log C &= \beta_5 + \alpha_5 \cdot \log s. \end{aligned} \tag{4}$$

Since the mean quantities $\langle n' \rangle$ and $\langle c' \rangle$ are given as $\frac{N}{s}$ and $\frac{C}{s}$, we can express the total production quantities N and C as $s \cdot \langle n' \rangle$ and $s \cdot \langle c' \rangle$ respectively. Dividing both sides of (3) by the team size s yields the following relation

$$\begin{aligned} \alpha_4 &= 1 + \alpha_2 \\ \alpha_5 &= 1 + \alpha_3 \end{aligned} \tag{5}$$

between the scaling factors of the models. As such, *negative scaling factors α_2 and α_3 in the models for the mean production* $\langle n' \rangle$ and $\langle c' \rangle$ *correspond to exponents α_4, α_5 smaller than 1 in the Cobb-Douglas production functions for the total production N* and C. In particular, such exponents smaller than 1 indicate a regime of *decreasing returns to scale*. In our case, we find values of $\alpha_4 = 0.25$ and $\alpha_5 = 0.14$, which provides further evidence to the fact that Open Source Software development processes are indeed examples of diseconomies of scale. Importantly, we emphasize that this finding is not an artifact of aggregating the data of all projects. Section A3 in the Appendix shows that it holds also on the level of individual projects.

From this we conclude that Open Source Software communities are indeed no magical exceptions from the basic economics of collaborative software engineering. Not surprisingly, we find that the productivity of large teams of irregularly contributing volunteers experiences a significant decrease as projects grow in size. As such, at least for productivity in OSS projects, it is safe to refute the Aristotelian interpretation that “the whole is more than the sum”, finding - on the contrary - strong support for the Ringelmann effect.

5 A Network Perspective on Coordination in Software Development

Considering our confirmation of the *Ringelmann effect* in OSS communities outlined above, we will now go one step further and investigate possible explanations for the observed relations between team size and productivity. In Section 3 we have argued that research has generally highlighted two different aspects that contribute to the Ringelmann effect: The first are motivational factors such as “free-riding” or “social loafing” which tend to become stronger as teams grow in size. The second are coordination challenges that naturally emerge as teams become larger. In line with a substantial body of work in empirical software engineering, our study exclusively focuses on coordination challenges emerging in development teams. While motivational factors are likely to play an important role as well, we argue that our available data do not allow us to quantitatively assess these factors.

In recent years, a number of sophisticated approaches have been developed which allow to operationalize coordination requirements in software development teams. Studying data from a software development repository, Cataldo et al. (2006) generated a so-called *Task Assignment Matrix* which captures the association between developers and the files they have modified. Using an additional data set of modification requests which can be used to identify interdependencies between files, the authors were then able to generate a *Coordination Requirement Matrix*. This matrix captures the need for coordination between developers who are either i) modifying the same files, or ii) editing different files that depend on each other due to a joint occurrence in the same modification request. Building on the general idea of operationalizing coordination requirements, researchers in empirical software engineering have generally highlighted the importance of fine-grained data on collaboration events in software development. Using rich data from the team collaboration platform JAZZ, Wolf et al. (2009) generated and analyzed developer communication networks in order to identify coordination needs and improve knowledge management. Recent works, have even gone beyond analyzing the mere presence of coordination needs, using time-stamped and fine-grained data of developer actions from support tools like MYLIN to not only detect (Blincoe et al. 2012), but also categorize coordination requirements into critical and non-critical ones (Blincoe et al. 2013; Blincoe 2014).

We acknowledge the importance of such accurate methods to infer meaningful coordination requirements based on fine-grained data that capture developer communication and developer actions beyond mere source code changes. However, in this paper we decided to follow a simpler approach for multiple reasons: First, we do not have access to fine-grained data on developer actions captured by collaboration tools such as MYLIN or JAZZ. Secondly, the fact that our study is comprised of 58 rather heterogeneous projects written in a total of 11 different programming languages, does not easily allow us to automatically extract comparable and meaningful dependency structures at at the level of programming language abstractions. And finally, rather than focusing on the microscopic inference of coordination requirements between a particular pair of developers, the goal of our study is a statistical categorization of team coordination structures at the macroscopic level.

In this paper we thus take a rather simplistic approach which is in essence a variation of the method proposed by Cataldo et al. (2006). In particular, using the file commit history of each project in our data set, we construct a Task Assignment matrix, that captures the association between developers and the source code regions they have edited. We particularly consider the time-stamps of commit actions, as well as the detailed code changes within all of the committed files in order to build directed and weighted networks which capture the

time-ordered co-editing of source code regions by different developers. We call the resulting network topologies *time-ordered co-editing networks* and throughout this article we will use them as a first-order approximation for the emerging coordination overhead.

5.1 Constructing Time-Ordered Co-editing Networks

We first provide a detailed description of our methodology of constructing time-ordered co-editing networks, which is further illustrated in Fig. 8. Let us denote each committed change of a single source code line l in file f at time t by developer A by the 4-tuple $\{A, f, l, t\}$. We infer a time-ordered co-editing relation $A \rightarrow B$ between developer A and B on file f, whenever developer A commits a change to a source code region of a file f, which was last edited by developer B. More formally, we infer a directed link (A, B) in a co-editing network whenever there are committed line changes $\{B, f, l_1, t_1\}$ and $\{A, f, l_2, t_2\}$ such that $t_1 < t_2$ and $l_1 = l_2$. We further require that no other developer X has edited this particular line in the meantime, i.e., we require $\nexists(X, f, l_1 = l_2, t')$ for some developer $X \neq B$ and some time stamp $t' \in [t_1, t_2]$. In order to be able to differentiate between trivial and more complex changes in the code, we further weight directed links by the size of the overlap of the co-edited code regions. Using the notation of committed lines from above, we do so by aggregating multiple links (A, B), each of which indicates a single co-edited line, to a weighted link (A, B, w), where the weight w captures the number of co-edited lines within a given productivity time window.

The method outlined above is enabled by a fine-grained analysis of the contents of all commits to all files in the full history of a project. In particular, we calculate *diffs* between two committed versions of a file whenever commits to this file occurred from different developers within a given productivity time window. For the technical details of our approach, we refer the reader to Section A6 in the Appendix. In the following we illustrate our method to construct time-ordered co-editing networks using the time-stamped commit activities shown in Fig. 8. In this example, three developers A, B and C are committing their changes to a total of three files, illustrated by differently colored document symbols

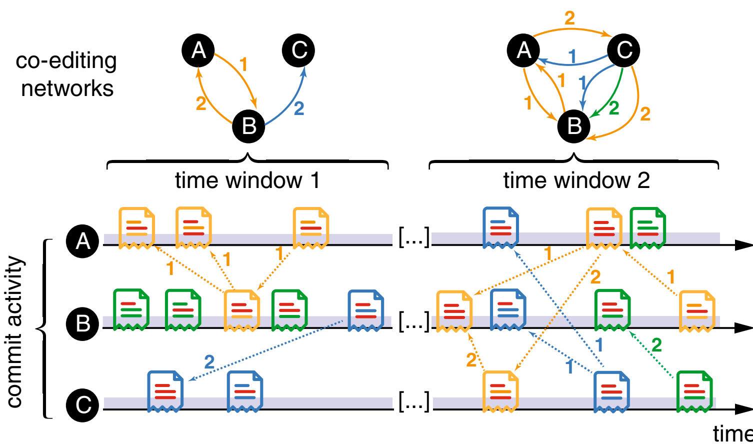

Fig. 8 Extraction of co-editing networks (*top panel*) from the commit activity (*bottom panel*) of three developers (A, B and C) editing lines (highlighted in *red*) in three different source code files (*orange, green and blue*)

in the time line of commit activities shown in the bottom panel. For simplicity here each of the files contains exactly three lines, and the lines edited in each of the commits are highlighted in red. In the left part of Fig. 8 (showing commit activities during time window 1) the sequence of code changes of developers is such that three time-ordered co-editing links emerge: developer B is connected to A since B edited two lines in the orange file which were previously edited by A. The fact that the overlap in the edits comprises two lines is reflected by an aggregate link weight of two in the co-editing network. Furthermore, developer A is connected to B, since A has edited line two in the orange file, which was last edited by developer B. Finally, developer B is connected to C by a directed link with a weight of two, since B has committed changes to lines one and three in the blue file, both of which were last changed by developer C. Importantly, unrelated code changes occurring in between the changes of two developers to the same lines will not result in a link in the co-editing networks. This can be seen in time window 2 shown in the right part of Fig. 8. Here, we infer a directed co-editing link with weight two from node C to node A, which is based on the changes to lines one and two in the green file. Notably, the intermediate change of line three in the green file by developer A does not result in a co-editing relation, even though it occurs in between the changes by B and C.

Compared to much more sophisticated methods to infer coordination requirements which have been outlined above, it is clear that our method merely generates an approximation for real coordination structures. Nevertheless we argue that the questions (i) which source code regions were changed by which developers, and (ii) in which order these changes occurred, capture interesting aspects of the team organization that influence potential coordination overhead emerging in a team. Indeed, much of the project management and architectural design efforts in software engineering aim at maintaining a *modular design*, which allows subcomponents of a system to be developed by specialized subgroups of developers, rather than arbitrary team members. The resulting partitioning of development tasks is meant to keep coordination requirements, and thus overhead, at a minimum level and to ensure that coordination requirements correspond to actual social structures (Cataldo et al. 2008). The association between edits to source code regions and developers captured by our time-ordered co-editing networks can be seen as a first-order approximation for the partitioning of development tasks.

To illustrate this, in the right panel of Fig. 8 we randomly shuffled the association between source code and developers in the commit history. As a consequence, each of the three files is now edited by each of the three developers, with no particular assignment of developers to certain subsets of files. The time-ordered co-editing network resulting from the commits in time window 2 thus becomes more densely connected. We interpret such a densification of co-editing networks as indicator for task assignment procedures that are likely to result in increasing coordination overhead. In particular, this densification implies a super-linear growth in the number of links as the number of nodes increases. In line with arguments about the super-linear increase in the number of possible communication channels which was quoted as one possible reason for *Brooks' law* (Brooks 1975), it can be seen as one possible explanation for decreasing productivity with increasing team size.

The fact that we study directed links whose directionality is defined based on the *order of committed changes*, further allows us to differentiate between nodes that *point to* many other nodes and nodes that are *pointed to by* many other nodes. Intuitively, by this we capture whether i) a developer A has to build on changes by many other developers (corresponding to a large *out-degree* of A), or ii) whether many other developers have to build on the changes committed by developer A (corresponding to a large *in-degree* of A). In addition, we consider *weighted links* between pairs of developers which can either result from

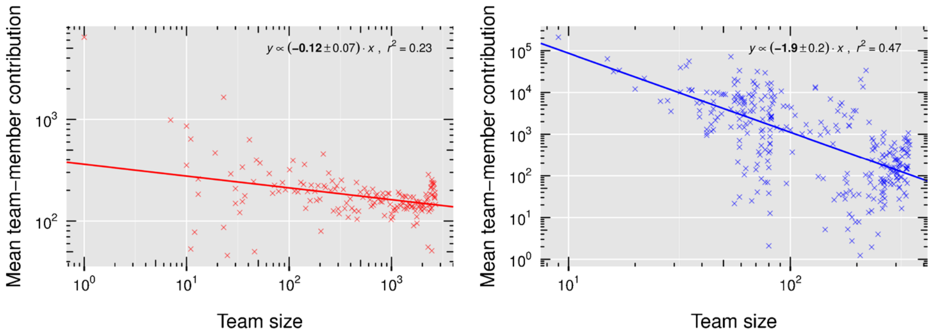

Fig. 9 Mean commit contribution per team member, depending on the team size for the projects SPECS (*red*), ZF2 (*blue*)

multiple co-edited lines on the same file or from co-edits on different files (see examples in Fig. 8). Considering weights in the resulting network topologies thus allows us to differentiate between weaker or stronger coordination links, based on the actual source code changes committed.

In summary, the approach outlined above allows us to capture both the topology and strengths of co-editing links between developers. A macroscopic characterization of network density allows us to quantitatively assess a proxy for potential coordination overhead introduced by the association between developers and edited source code regions. Building on *Brooks' law*, we hypothesize that co-editing network becomes increasingly dense as the number of active developers grows. In other words, we hypothesize that the *mean* number of co-editing relations introduced per team member grows as the number of developers increases.

5.2 Co-editing Networks and Productivity

We first test this hypothesis by means of a case study, focusing on two specific large-scale projects covered by our data set. For the selection of these projects, we computed the coefficients α_3 of a log-transformed linear model for the relation between the team size and the mean commit contribution per team member (see Section 4.1 and (2) for details of the model). Different from Section 4.1, here we perform this regression analysis *per project*, while limiting ourselves to those 15 projects in our data set which have at least 10,000 commits (Table 3). In Table 4 in the Appendix we report on the analysis of the fits. In order to be able to compare the scaling exponents of these projects to each other, we further limit our analysis to those 11 of the 15 large-scale projects, for which the log-transformed linear model from (2) provides the most reasonable fit (see Section A3 in the Appendix for details).

From these 11 projects, we selected (i) ZF2 being the project with the *smallest* exponent $\alpha_3 \approx -1.9$, and (ii) SPECS being the project with the *largest* exponent $\alpha_3 \approx -0.12$.[9] Figure 9 shows the relation between the team size and the mean commit contribution per team member for these two projects individually.

[9]Notably, all of the 15 large projects exhibit *negative* coefficients, which indicate the presence of the Ringelmann effect.

Using the methodology described above, we next create time-evolving co-editing networks for the two projects SPECS and ZF2 as outlined above. In Fig. 10 three representative snapshots of the resulting networks are shown for each project. For these networks, we are now interested in the question of how the co-editing links per developer scale with the size of the network. Since we are using weighted networks, there are different ways of computing the co-editing links per developer. First, for each developer A, we can simply sum the weights of all outgoing links (A, X). This *weighted out-degree* captures the total number of co-edited lines, which can be seen as a proxy for coordination costs incurred by developer A. Secondly, we can discard weights which capture the *number* of co-edited lines between a particular pair of developers, instead solely focusing on the *topological dimension*. For this, we compute the (unweighted) out-degree of each developer A, capturing with *how many* different other developers a developer A has co-edited source code regions. In Fig. 11 we show the relation between the size of the co-editing networks, computed in terms of the number of nodes, and the mean coordination costs per developer, proxied by the different degree-based measures outlined above.

For the ZF2 project, which exhibits the *smallest* coefficient α_3 in our data set, we observe a positive correlation between the size of the network and the mean (weighted and unweighted) out-degree. We quantify this positive correlation using a linear regression analysis. Precisely, we calculate the coefficient of a linear model for the growth of the mean degree. Since the mean degree in a network is bounded above by the the number of nodes, this coefficient is necessarily smaller than one. At the same time, a coefficient of zero would indicate a constant mean degree, and thus a linear growth of the number of links in the network. For the growth of the mean (unweighted) out-degree in the ZF2 project, we obtain a coefficient of 0.031 with an associated r^2 value of 0.23. For the mean weighted out-degree,

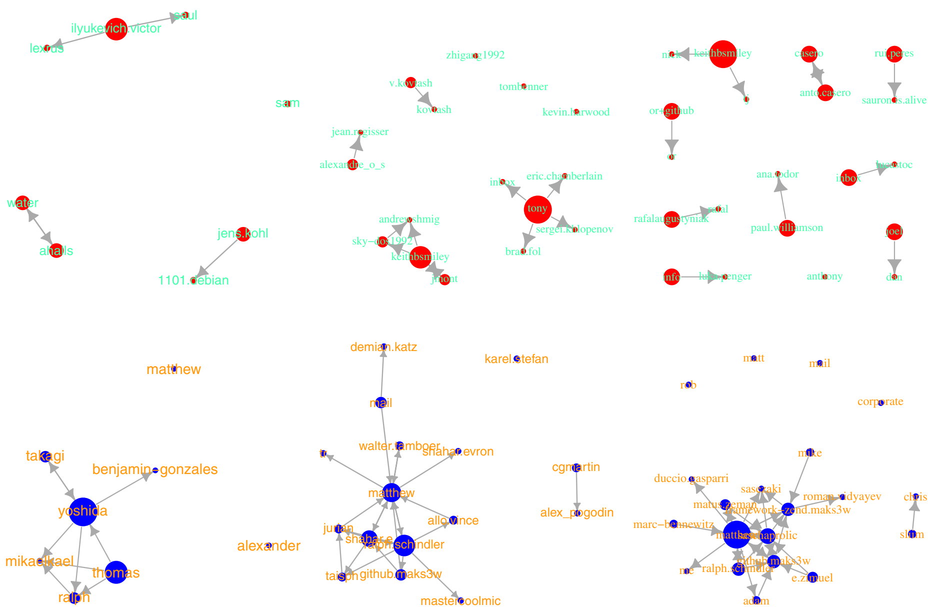

Fig. 10 Six representative coordination networks of (i) the SPECS project (*red* nodes) with sizes of 8, 16 and 20 developers, and (ii) the ZF2 project (*blue* nodes) with sizes of 8, 16 and 20 developers. The identity of the developers is displayed as their email address (without domain name)

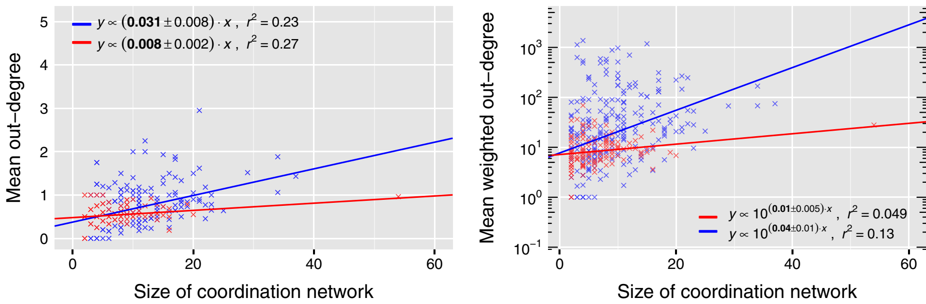

Fig. 11 Mean degree versus size of coordination networks for the projects ZF2 (*blue*) and SPECS (*red*)

we use a linear regression model on log-transformed values for the weighted out-degree. We then calculate the scaling coefficient for the exponent of a log-linear model. Again, a zero coefficient would indicate a linear growth of the number of links, while any positive coefficient indicates a super-linear growth. For the ZFS project, we obtain a coefficient of 0.04 with an associated $r^2 = 0.13$. These positive values of the coefficients are evidence for a super-linear growth of co-editing links as the number of developers increases, which can be interpreted as a growth in the *mean* coordination cost per developer. The resulting tendency of networks in the ZF2 project to become increasingly dense can be observed in the bottom panel of Fig. 10.

For the SPECS project, which exhibits the *largest* coefficient α_3 in our data set, we observe a significantly less pronounced growth of the (weighted and unweighted) degree as the network size grows. A regression analysis for the mean (unweighted) out-degree yields a coefficient 0.008 with an associated r^2 value of $r^2 = 0.27$, while for the mean weighted out-degree we obtain a coefficient 0.01 and $r^2 = 0.049$. From this, we conclude that an increase in the number of developers in the SPECS project results in a much less pronounced growth of co-editing links, compared to the ZF2 project. The resulting, rather sparsely connected networks in the SPECS project can be observed in the top panel of Fig. 10. This finding provides a possible explanation for the scaling exponent of $\alpha_3 \approx -0.12$ which, compared to all other projects, corresponds to rather slowly decreasing returns to scale. Conversely, the fast, super-linear increase of co-editing links in the ZF2 project coincides with a particularly small scaling exponent of $\alpha_3 \approx -1.9$ which corresponds to rapidly decreasing returns to scale. As such, the two case studies are consistent with our hypothesis that the decreasing returns to scale in OSS communities are related to an *increasing* densification of co-editing networks.

A closer look at the two projects in our case study allows us to further substantiate our quantitative findings with project-dependent, intuitive explanations. The SPECS project maintains a public repository for meta-information on APPLE COCOA modules, that can be managed via the library dependency management tool COCOAPODS. In particular, the repository consists of more than 7000 different libraries, each containing a set of JSON files which are authored and maintained by different, small teams of developers. As such, SPECS can be seen as a rather extremal example for a *maximally modular project*, in the sense that it consists of thousands of independent modules each being developed by different developers with no or at least minimal coordination needs.

The ZF2 project on the other hand, develops a component-oriented web framework named *Zend*. Compared to the fully independent modules in the SPECS project, the code base of ZF2 is likely to be much more integrated, thus not allowing for a non-overlapping

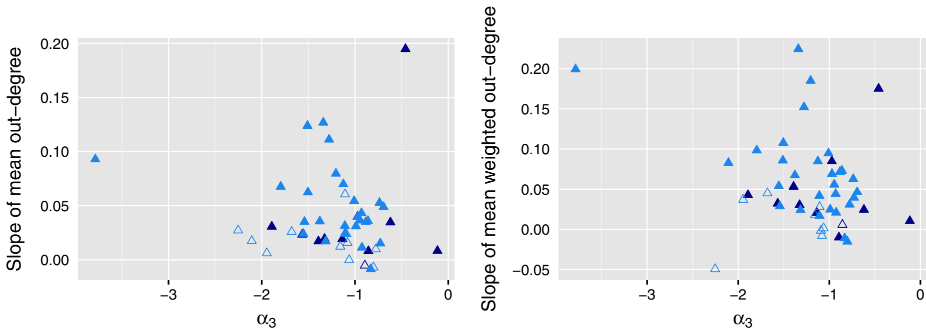

Fig. 12 Scaling of the mean team-member contribution with team size (α_3) vs. the scaling coefficients of the mean out-degree and mean weighted out-degree with the size of the coordination networks. To make projects comparable, we selected 48 of the 49 projects for which the log-log model is the *best* and *significant* fit for the scaling of productivity. We removed one project, OH-MY-ZSH, for which we could not estimate a reliable slope of the mean-degree. The coordination network for OH-MY-ZSH was so sparse that our robust regression did not converge. Filled symbols indicate that the fitted scaling coefficients for the mean (weighted) out-degree in the network is significant at $p = 0.05$. *Dark blue* symbols represent those projects which have at least 10,000 commits

mapping between developers and source code regions. As such, the commit actions of a large number of developers are likely to produce a much more densely connected co-editing network. A particular additional feature of the ZF2 project is that it imposes rather rigorous standards with respect to programming style, documentation and testing which are summarized in a detailed *contributor guide*.[10] These standards are targeted at improving both maintainability and software quality and require, for instance, that any piece of contributed code must be covered by unit tests. This focus on rigorous coding standards could be one possible explanation for an increased coordination overhead and thus a significantly smaller scaling exponent α_3, while at the same time possibly translating to increased software quality and maintainability.

In addition to this case study focusing on the two projects with the smallest and largest decrease in productivity, we additionally perform an analysis for all 48 projects, for which the log-log model is the best and significant fit for the scaling of productivity. The results of this analysis are shown in Fig. 12. Here, the scaling exponent α_3 for the scaling of productivity is shown on the x-axis, while the coefficient for the slope of the mean out-degree and the mean weighted out-degree are shown on the y-axis of the left and the right panel respectively. Dark blue symbols refer to projects with at least 10,000 commits, filled symbols indicate projects for which the fitted scaling coefficient for the mean (weighted) out-degree is significant at $p = 0.05$. The results highlight that all projects with strongly negative and significant slopes α_3 for the scaling of productivity also exhibit pronouncedly *positive* scaling exponents for the growth of the mean (weighted) out-degree. Based on these empirical results, we can thus not reject our hypothesis that the decrease in productivity for larger team sizes is related to a super-linear growth of coordination links. While the spread in Fig. 12 does not indicate the presence of a strong linear correlation, one nevertheless observes a general tendency of projects with larger (i.e., closer to zero) scaling exponents α_3 to exhibit smaller coefficients for the growth of the mean out-degree. We can quantitatively assess the significance of such a relation by computing the normalized mutual information (NMI)

[10]see https://github.com/zendframework/zf1/wiki/Contributing-to-Zend-Framework-1

between the slopes of the mean (weighted) out-degree and the exponents α_3. This quantity measures the extent to which knowing the value of one variable allows the inference of the other. The NMI has a minimum of 0 when the two variables are independent, and a maximum of 1 when one variable completely determines the other. Referring to the left side of Fig. 12, the NMI between α_3 and the slope of the mean out-degree is 0.33. Similarly on the right side, the NMI between α_3 and the slope of the mean weighted out-degree is 0.24.

We further estimate the statistical significance of the obtained NMI values by means of a bootstrapping approach. We shuffle α_3 and the mean (weighted) out-degree 10^4 times and compute the resulting NMI for each shuffled sample. Calculating the percentage of the shuffled NMI values that are larger than the empirical ones allows us to derive a p-value for the null hypothesis that the empirical NMI values of 0.33 and 0.24 can be obtained by chance alone. The calculated p-values are 0 and 0.009 for the left and right side of Fig. 12 respectively, which allows us to reject the null hypothesis above. We therefore conclude that the positive NMI values are significant, and thus substantiate the negative relation between the scaling exponents α_3 and the growth coefficients of the mean (weighted) out-degree.

6 Threads to Validity and Future Work

Prior to concluding our article, in the following we summarize a number of threats to validity and open issues that highlight potential directions for future research.

First, and foremost, our study is based on a large-scale data set from the social coding platform GITHUB which comes, like any data set, with its own fallacies and potential biases. The specific risks and opportunities awaiting researchers in the analysis of data from GITHUB have recently been summarized by Kalliamvakou et al. (2014). A number of risks are associated with the facts that i) GITHUB repositories are not necessarily equivalent to projects, ii) the majority of repositories show low levels of commit activity, iii) the majority of repositories are personal repositories not used for collaboration, iv) many repositories are not related to software development, and v) many GITHUB repositories are actually not the main development repository of projects. We avoided all of these risks by a careful selection of projects which ensures that all analyzed repositories i) are the *main* development repositories of OSS projects, ii) showed activity for at least one year, iii) showed recent activity in the week of data selection, and iv) had commits by at least 50 different developers. A second set of risks involves the use of so-called *pull requests*, and the way in which those are recorded in the commit history. As mentioned in Section 3.1, by our special treatment of merged pull requests we have carefully accounted for the potential resulting biases. In particular for merged pull requests we only account for the contribution of the original commit underlying the pull request. We further correctly attribute this contribution to the actual author of the commit, and not to the maintainer who merged the pull request (see details in Section A2) .

One shortcoming of our empirical analysis is that we have excluded motivational factors which have been shown to be important driving factors of the Ringelmann effect. We expect that such factors play an important role as a further mechanism behind the decreasing returns to scale observed in our study, and we thus highlight a large potential for future research in this area. A particularly interesting question that results from our choice of data from GITHUB, is whether the interaction mechanisms of this platform may mitigate motivational factors contributing to the Ringelmann effect. In particular, a recent study has highlighted the importance of publicly available developer activity information on GITHUB (Dabbish et al. 2012), which is supposed to have a significant effect on the motivation of

developers. Building on our quantitative approach, future work may search for quantitative evidence for this effect, for instance by means of a large-scale comparison between the scaling of team productivity on GITHUB and other platforms such as, e.g., SOURCEFORGE. In order to facilitate such future work, we have made both the raw and the processed version of our data set freely available to be reused for researchers (Scholtes et al. 2015).

One concern about the validity of our results on the scaling exponents of productivity may arise due to our estimation of the size of development teams. To define a reasonable size of the time-window for our analysis, we used the statistics of developer activity, i.e., the maximum time of inactivity between any two commits of a developer. Notably, one-time contributors, i.e., developers who committed only once, were not included in the statistics of inactivity in order to avoid a bias towards large inactivity times. However, the question whether the large fraction of one-time contributors for some projects may result in a bias needs to be addressed. On the one hand, a large number of one-time contributors with minor commit contributions could result in a spuriously large team size that results in an underestimation of productivity scaling exponents. On the other hand, even though the fraction of single-commit developers is large for some projects, by definition these developers committed only once, thus limiting the magnitude of the effect. While there is no obvious answer to the question of how to account for single-commit contributions and whether to count one-time contributors as members of a development team, we carefully avoided potentially wrong conclusions in two ways: First, in Section 4, we confirmed a sub-linear scaling of productivity both for the size of the *team* and the number of *active developers*. The latter number is defined using a time window of 7 days, which significantly mitigates potential biases that may affect an analysis based on the much larger team size window. Secondly, to avoid wrong conclusions based on our results on the effect of the team size, we reanalyzed all of our data focusing on a stricter notion of *core developers* which excludes all *one-time contributors*. We report these results in Section A4 both at the aggregate and the project level. The scaling exponents shown in Tables 5 and 6 confirm that one-time contributors do not significantly influence our results and that they do not qualitatively change any of our conclusions.

Finally, a further limitation of our work is our rather simplistic perspective on coordination structures in software development teams. As discussed in Section 5, this perspective was chosen due to the available repository data, the heterogeneity of the projects investigated, as well as the macroscopic approach which is the focus of our study. An interesting extension of our work is thus to relate our results on the scaling of team productivity with more meaningful and fine-grained abstractions for coordination requirements, like those investigated in Cataldo et al. (2008), Wolf et al. (2009), and Blincoe et al. (2012, 2013). Such a study would need to combine data on a sufficiently large set of projects with detailed data from task repositories. A second approach to extend our study could be the focus on projects developed in particular programming languages which would allow for an automated extraction of dependency networks, and thus a more reasonable notion of coordination requirements that goes beyond the co-editing of source code regions.

7 Conclusion

In the following, we conclude our article by summarizing our main findings and contributions.

Using a data set covering more than 580,000 commits from 58 Open Source Software projects hosted on GITHUB, we first investigated reasonable measures of productivity which can be defined based on commit log data. In addition to studying the number of commits, we take into account the distribution of *commit contributions* measured in terms of the Levenshtein distance between the version of source code files before and after a commit. Using a time-slice analysis, we first showed that the mean contribution of commits exhibits a large variance over several orders of magnitude. We argue that - due to this large variance - reasonable quantitative proxies for productivity need to take into account the actual contribution of commits rather than being based on their mere number.

Using a notion of productivity that encompasses both the number of commits, as well as the contributions of those commits in terms of source code changes, we studied the relation between the size of the development team and the productivity of team members. We find strong evidence for a negative relation between team size and productivity. Through a log-transformed robust linear regression analysis, we further computed a scaling exponent for the relation between productivity and team size. Our results confirm negative scaling exponents both for the aggregated data containing all projects, as well as for all of the 58 projects individually. With this, our study provides quantitative evidence for the presence of the Ringelmann effect in Open Source Communities. By means of the expression of the observed scaling relations in terms of a Cobb-Douglas production function, we further conclude that all of the studied projects represent *diseconomies of scale*, exhibiting diminishing returns to scale.

Investigating possible explanations for the decrease in productivity as teams grow in size, we have taken a network perspective in order to study the growth of coordination overhead. Based on the association between developers and committed source code regions, as well as the timing and ordering of these commits, we infer directed and weighted networks which capture the co-editing of source code regions between developers. We performed a case study on two extremal projects in our data set which show the most and the least pronounced decrease in productivity as teams grow larger. For these two projects we were, through a detailed analysis of the time-evolving co-editing networks, able to confirm our hypothesis that the scaling of productivity is related to the growth dynamics of links in the coordination network. Extending this analysis to all 49 projects for which we could infer a significant fit to the scaling of productivity, we further show that i) all projects with a substantially negative scaling of productivity exhibit a strong super-linear growth of links in co-editing networks, ii) there is a statistically significant relation between the growth of co-editing networks and the scaling coefficient of productivity.

In summary, our results confirm the intuition that - due to the overhead of coordinating the work of a large number of developers - large-scale Open Source Software projects are examples for diseconomies of scale. As such, we find that OSS communities are no exceptions from basic software engineering economics, which suggests the emergence of negative non-additive effects in increasingly large teams of collaborating developers. We thus conclude that it is safe to reject the presence of synergetic effects which have been associated with the Aristotelian quote that "the whole is more than the sum of its parts". We instead find that, like other collaborative software engineering projects, OSS communities are examples for collaborative human endeavours exhibiting a pronounced Ringelmann effect.

Acknowledgments Ingo Scholtes and Frank Schweitzer acknowledge support from the Swiss National Science Foundation (SNF), grant number CR31I1_140644/1.

Appendix A

A1 Data Set

Table 3 summarises the 58 projects in our data set. For each project we show the project name and programming language, the time span of the data retrieved, as indicated by the times of the first and last retrieved commits, the total number of commits and the total number of unique developers during the analysed time span.

Table 3 Summary of the 58 OSS projects in our data set

Project	Language	From	To	Commits	Committers
antirez/redis	C	2009-03-22 09:30:00	2014-10-29 11:48:22	4361	173
mono/mono	C#	2001-06-08 18:45:34	2014-11-21 14:20:40	96688	738
xbmc/xbmc	C++	2009-09-23 01:49:50	2014-10-30 13:39:03	25810	543
TrinityCore/TrinityCore	C++	2008-10-02 21:23:55	2014-10-30 06:38:55	22275	487
cocos2d/cocos2d-x	C++	2010-07-06 02:19:51	2014-10-30 04:45:22	16400	448
Itseez/opencv	C++	2010-05-11 17:44:00	2014-10-28 13:06:36	11972	383
bitcoin/bitcoin	C++	2009-08-30 03:46:39	2014-10-30 06:31:34	4624	314
dogecoin/dogecoin	C++	2009-08-30 03:46:39	2014-08-24 14:57:20	4036	269
litecoin-project/litecoin	C++	2009-08-30 03:46:39	2014-09-16 09:50:58	2935	190
twbs/bootstrap	CSS	2011-04-27 20:53:51	2014-10-30 16:13:39	7461	670
zurb/foundation	CSS	2011-10-13 23:09:47	2014-10-28 23:54:09	5525	676
docker/docker	Go	2013-01-19 00:13:39	2014-10-29 23:43:18	7216	752
elasticsearch/elasticsearch	Java	2010-02-08 13:30:06	2014-10-30 11:32:53	9838	378
libgdx/libgdx	Java	2010-03-06 16:05:53	2014-10-30 12:56:30	7955	332
github/android	Java	2011-10-12 22:36:58	2014-08-08 11:09:08	2305	67
jquery/jquery-mobile	JavaScript	2010-09-10 22:23:13	2014-10-29 23:13:23	10847	285
meteor/meteor	JavaScript	2011-11-18 02:35:20	2014-07-25 20:57:47	8162	146
adobe/brackets	JavaScript	2011-12-07 21:20:16	2014-10-30 15:35:15	8068	232
mrdoob/three/js	JavaScript	2010-04-24 03:01:19	2014-10-28 22:33:52	7557	440
joyent/node	JavaScript	2009-02-16 00:02:00	2014-10-02 16:00:40	6918	510
jquery/jquery-ui	JavaScript	2008-05-22 15:38:37	2014-10-25 16:18:17	6146	285
angular/angular/js	JavaScript	2010-01-06 00:36:58	2014-10-30 15:04:41	5930	1147
jquery/jquery	JavaScript	2006-03-22 03:33:07	2014-10-30 13:16:32	5376	249
emberjs/ember/js	JavaScript	2011-04-30 22:39:07	2014-10-30 12:54:26	5078	504
ajaxorg/ace	JavaScript	2010-04-02 13:39:45	2014-10-30 00:09:09	4695	241
mozilla/pdf/js	JavaScript	2011-04-26 06:33:36	2014-10-28 18:56:55	4630	201
strongloop/express	JavaScript	2009-06-26 18:56:18	2014-10-29 05:15:58	4459	183
cocos2d/cocos2d-html5	JavaScript	2012-01-29 09:14:21	2014-12-17 03:33:02	3468	93
mbostock/d3	JavaScript	2010-09-27 17:23:59	2014-10-23 17:05:38	2640	86
tryghost/Ghost	JavaScript	2013-05-04 11:09:13	2014-12-15 17:09:24	2529	210
jashkenas/backbone	JavaScript	2010-09-30 19:48:05	2014-10-28 01:52:45	1938	265
tastejs/todomvc	JavaScript	2011-06-03 20:04:08	2014-10-30 00:58:23	1491	226
ivaynberg/select2	JavaScript	2012-03-04 18:58:26	2014-10-29 21:36:54	972	316

Table 3 (continued)

Project	Language	From	To	Commits	Committers
AFNetworking/ AFNetworking	Objective-C	2011-05-31 21:27:34	2014-10-25 02:34:33	1579	253
WordPress/WordPress	PHP	2003-04-01 06:17:43	2014-10-16 22:07:20	27101	55
zendframework/zf2	PHP	2009-04-28 10:23:49	2015-01-13 10:14:03	16810	890
symfony/symfony	PHP	2010-01-04 14:26:20	2014-10-27 18:25:03	12945	1236
cakephp/cakephp	PHP	2005-05-15 21:41:38	2014-10-30 01:43:18	12224	349
bcit-ci/CodeIgniter	PHP	2006-08-25 17:25:49	2014-10-29 11:18:24	5987	372
laravel/laravel	PHP	2011-06-09 04:45:08	2014-09-29 14:08:27	3155	264
yiisoft/yii	PHP	2008-09-28 12:03:53	2014-10-24 15:29:38	2415	220
django/django	Python	2005-07-13 01:25:57	2014-10-30 12:53:20	18266	798
ansible/ansible	Python	2012-02-05 17:48:52	2014-10-29 04:59:46	8825	1131
kennethreitz/requests	Python	2011-02-13 18:52:37	2014-10-26 13:45:12	2778	371
mitsuhiko/flask	Python	2010-04-06 11:12:57	2014-10-27 10:54:30	1629	270
rails/rails	Ruby	2004-11-24 01:04:44	2014-10-30 16:01:18	39509	3046
CocoaPods/Specs	Ruby	2011-09-08 19:46:11	2014-10-30 15:52:28	25412	4107
rapid7/metasploit-framework	Ruby	1970-01-01 00:02:02	2014-10-28 17:10:19	22354	387
spree/spree	Ruby	2008-02-25 18:23:50	2015-01-13 09:12:45	14171	742
diaspora/diaspora	Ruby	2010-06-11 17:40:49	2014-10-23 05:12:26	9773	360
gitlabhq/gitlabhq	Ruby	2011-10-08 21:34:49	2014-10-30 12:44:29	8991	690
fog/fog	Ruby	2009-05-18 07:13:06	2014-10-30 12:08:47	8548	752
discourse/discourse	Ruby	2011-10-15 18:00:00	2014-10-30 16:11:33	7026	338
mitchellh/vagrant	Ruby	2010-01-21 08:35:06	2014-10-25 14:10:37	3210	322
activeadmin/activeadmin	Ruby	2010-04-15 13:23:16	2014-12-15 22:00:26	2754	433
plataformatec/devise	Ruby	2009-09-16 12:17:43	2014-10-29 14:59:33	2368	414
jekyll/jekyll	Ruby	2008-10-20 02:07:26	2013-06-08 17:47:34	1486	210
robbyrussell/oh-my-zsh	Shell	2009-08-28 18:14:03	2014-10-22 13:16:15	1732	796

A2 GitHub Data Set

For each project in our data set we queried the GITHUB API with the query https://api.github.com/repos/<owner>/<repo>/commits?page=<n>, where <owner> is a GITHUB user account and <repo> is the name of a project repository belonging to this user. The query returns a paginated json list of the 30 most recent commits in the master branch of the project. By varying the parameter <n>, we control the pagination and can trace back the commit history until the very first commit.

Each element in the json list represents a commit with all GIT-relevant information (see Section 3.1). More specifically, it contains the names and email addresses of both the author and the committer.[11] The author is the person who authored the code in the commit and the committer is the one with write permissions in the repository who merged the commit in the project code base. These two identities may not be the same when pull requests are

[11] Developers who use Git input this basic information in the configuration of their Git clients.

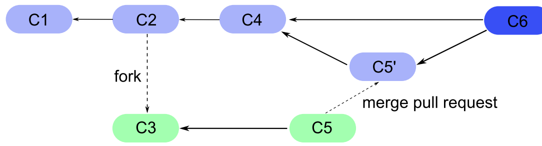

Fig. 13 Simplified illustration of merging a pull request. A potential contributor forks the main branch of a project (*light blue*) into his/her own local repository (*green*). After some activity in both repositories a pull request is created and merged as indicated by the *dashed arrow*. This results in two commits - C5' and C6 - in the main branch. The merge commit C6 (*dark blue*) has two parent links and should be excluded

considered, as the developers requesting the pull typically do not have write access. Since we quantify contributions in terms of amount of code written, we take the author email from the commit data as a unique identifier for individual developers. In cases where the author email is empty, we conservatively skip the commit.

The commit SHA contained in the json list can be used to execute a commitspecific query in the GitHub API of the form:

https://api.github.com/repos/<owner>/<repo>/commits/<SHA>

The result is again a json list which provides detailed information about the list and diffs of all files changed in the commit. We retrieve this additional information and use it to (i) quantify the precise contribution to the source code at the level of individual characters and (ii) construct the time-varying coordination networks of developers who have co-edited files (see Section 5.1).

A2.1 Merged Pull Requests

Upon merging a pull request, typically through the GitHub interface, the commit tree of the project is modified by including a special merge commit. The basics of this process is illustrated in Fig. 13.

In this example, a potential contributor forks the main branch after the second commit. Subsequent local changes are then made to the master branch and to the remote repository, represented by commits C4 and C5 respectively. After C5, the potential contributor creates a pull request asking for the changes in C5 to be incorporated in the main code base. Assuming the pull request is approved and no conflicts exist, C5 is merged by creating two commits - C5' and C6. C5' is almost identical to C5 in that it has the same author and committer fields as well as diffs.[12] C6 is a special merge commit that contains the same diffs as C5 and C5, but differs on the author and committer information. The author and committer in C6 are those of the maintainer who approved and merged the pull request, and not those of the developer who originally wrote the code in C5 and C5'. Thereby including commit C6 in the analysis would wrongly attribute the contained diff to the maintainer and inflate his/her contribution in terms of code written.

We deal with this problem by noticing that merge commits always have at least two parent pointers - one to the replicated commit from the forked repository, and one to the

[12]The differences pertain to the hashes of the corresponding physical files, which are irrelevant for us.

last commit in the main branch. In some cases when changes are merged from more than one remote branches, the merge commit will have a parent pointer to each of these remotes. Since the parent pointers are also available in our data set, we exclude all commits that have two or more parent pointers.[13]

An additional complication is that GIT also allows integrating changes by socalled *rebasing*. Different from pull requests, which generate a merge commit, in rebasing all changes are applied on top of the last commit of the branch being rebased into. The result is a single commit with only one parent link that is added at the end of the rebased branch and that incorporates these changes. Since we cannot distinguish the developer who rebased from those who authored the changes, we exclude such commits from our analysis. Even though the parent pointer rule cannot be applied here, most well-structured projects contain indicative commit messages that can be used to this end. We exclude all commits with commit messages that contain any of the keywords merge pull request, merge remote-tracking, and merge branch, regardless of punctuation.

We note that all summary statistics regarding the number of commits in this paper (e.g. Table 3) are calculated after applying the above two exclusion methods.

A3 Model Fits for Project-Wise Scaling of Productivity

For each project in our data set, we estimated the model in (2) relating the team size s to the mean team-member contribution $\langle c' \rangle$. For a small number of those projects, the team size s varies in a rather narrow range, thus questioning logarithmic transformations of both the parameter s and $\langle c' \rangle$ in the linear model of (2).[14] We thus additionally use a model variation with a logarithmic transformation of $\langle c' \rangle$, while keeping s linear, i.e.:

$$\log\langle c' \rangle = \hat{\beta}_3 + \hat{\alpha}_3 \cdot s \tag{6}$$

We denote this model as *Log-Lin*, while referring to the original model in which we perform a logarithmic transformation of both parameters as *Log-Log*.

For each project, we fit both models and select the one which yields the largest coefficient of determination r^2 as the appropriate model for this project. The resulting project-dependent scaling coefficients are summarized in Table 4.

The result confirms that our finding of decreasing returns to scale at the aggregate level (Section 4.1) also holds for individual projects. Virtually all projects exhibit negative scaling of the mean team-member contribution with the team size, except for two projects for which no significant scaling coefficient could be determined. At any rate, the absence of significant positive coefficients for any of the projects allows us to conclude that there is no evidence for super-linear scaling in our data set.

[13] Note that this method is valid also when changes are merged from other local branches, and not from forked repositories. The corresponding merge commit still contains parent pointers linking it to these branches.

[14] Note that due to the sensitivity of the Levenshtein index, it varies across several orders of magnitude, regardless of the project size, hence a log transformation on $\langle c' \rangle$ is justified.

Table 4 Robust linear regression of the *Log-Log* and *Log-Lin* models for all projects

	Log-Log		Log-Lin		
Project	α_3	r^2	$\hat{\alpha}_3$	$\hat{r}^2$	
zendframework/zf2	−**1.892** (−**2.038**)	0.466 (0.462)	−0.006 (−0.008)	0.416 (0.440)	≥ 10, 000 commits
xbmc/xbmc	−**1.566** (−**1.701**)	0.330 (0.274)	−0.007 (−0.010)	0.304 (0.255)	
cakephp/cakephp	−**1.392** (−1.430)	0.717 (0.680)	−0.018 (−**0.024**)	0.715 (0.681)	
spree/spree	−**1.326** (−**1.351**)	0.504 (0.466)	−0.006 (−0.011)	0.311 (0.296)	
TrinityCore/TrinityCore	−**1.136** (−**1.151**)	0.814 (0.791)	−0.011 (−0.014)	0.684 (0.641)	
Itseez/opencv	−**0.972** (−**0.975**)	0.825 (0.799)	−0.012 (−0.017)	0.783 (0.763)	
rails/rails	−**0.894** (−**0.900**)	0.764 (0.714)	−0.003 (−0.004)	0.675 (0.620)	
django/django	−**0.856** (−**0.850**)	0.475 (0.360)	−0.004 (−0.006)	0.356 (0.257)	
cocos2d/cocos2d-x	−**0.620** (−**0.590**)	0.257 (0.232)	−0.004 (−0.005)	0.132 (0.132)	
WordPress/WordPress	−**0.459** (−**0.459**)	0.042 (0.048)	−0.017 (−0.019)	0.025 (0.032)	
jquery/jquery-mobile	−0.352 (0.260)	0.01 (0.004)	−0.001 (0.002)	−0.004 (−0.002)	
CocoaPods/Specs	−**0.116** (−**0.085**)	0.229 (0.117)	0 (0)	0.053 (0.006)	
mono/mono	−2.276 (−1.122)	0.37 (0.170)	−**0.013** (−**0.016**)	0.490 (0.284)	
rapid7/metasploit-framework	−0.511 (−0.48)	0.263 (0.22)	−**0.006** (−**0.006**)	0.288 (0.247)	
symfony/symfony	−1.085 (−1.099)	0.543 (0.484)	−**0.004** (−**0.006**)	0.57 (0.508)	
mitsuhiko/flask	−**3.785** (−**3.827**)	0.278 (0.181)	−0.051 (−0.093)	0.268 (0.148)	< 10, 000 commits
github/android	−**2.254** (−**2.861**)	0.486 (0.364)	−0.062 (−0.154)	0.477 (0.353)	
laravel/laravel	−**2.107** (−**2.290**)	0.406 (0.374)	−0.012 (−0.028)	0.196 (0.234)	
plataformatec/devise	−**1.946** (−**1.909**)	0.429 (0.269)	−0.024 (−0.035)	0.344 (0.136)	
jashkenas/backbone	−**1.796** (−**1.141**)	0.142 (0.059)	−0.008 (−0.011)	0.023 (0.014)	
AFNetworking/AFNetworking	−**1.679** (−**1.889**)	0.364 (0.264)	−0.018 (−0.051)	0.264 (0.206)	
ivaynberg/select2	−**1.554** (−**1.887**)	0.476 (0.443)	−0.013 (−0.058)	0.323 (0.380)	
discourse/discourse	−**1.542** (−**1.501**)	0.327 (0.335)	−0.004 (−0.008)	0.16 (0.187)	
yiisoft/yii	−**1.510** (−**1.638**)	0.531 (0.527)	−0.017 (−0.027)	0.298 (0.316)	
mozilla/pdf.js	−**1.505** (−1.584)	0.476 (0.386)	−0.022 (−**0.04**)	0.428 (0.399)	
activeadmin/activeadmin	−**1.378** (−1.54)	0.637 (0.538)	−0.016 (−**0.031**)	0.633 (0.565)	
jquery/jquery-ui	−**1.340** (−**1.87**)	0.267 (0.139)	−0.014 (−0.037)	0.173 (0.096)	
emberjs/ember.js	−**1.315** (−**1.399**)	0.462 (0.440)	−0.004 (−0.006)	0.189 (0.187)	
cocos2d/cocos2d-html5	−**1.278** (−**1.237**)	0.407 (0.314)	−0.031 (−0.037)	0.338 (0.238)	
tastejs/todomvc	−**1.205** (−**1.114**)	0.129 (0.084)	−0.016 (−0.024)	0.098 (0.052)	
mitchellh/vagrant	−**1.158** (−1.279)	0.438 (0.355)	−0.009 (−**0.027**)	0.429 (0.418)	
kennethreitz/requests	−**1.125** (−**1.212**)	0.245 (0.237)	−0.009 (−0.019)	0.188 (0.199)	
libgdx/libgdx	−**1.109** (−**1.121**)	0.496 (0.451)	−0.011 (−0.016)	0.391 (0.356)	
twbs/bootstrap	−**1.107** (−**1.108**)	0.293 (0.233)	−0.004 (−0.009)	0.138 (0.126)	
tryghost/Ghost	−**1.106** (−**1.136**)	0.549 (0.386)	−0.008 (−0.014)	0.382 (0.253)	
adobe/brackets	−**1.092** (−**1.095**)	0.500 (0.469)	−0.010 (−0.012)	0.350 (0.330)	
gitlabhq/gitlabhq	−**1.081** (−**1.085**)	0.589 (0.488)	−0.004 (−0.009)	0.523 (0.436)	
ansible/ansible	−**1.062** (−**1.079**)	0.667 (0.598)	−0.002 (−0.003)	0.395 (0.356)	

Table 4 (continued)

Project	Log-Log		Log-Lin	
	α_3	r^2	$\hat{\alpha}_3$	$\hat{r}^2$
litecoin−project/litecoin	**−1.009** (**−0.972**)	0.270 (0.202)	−0.016 (−0.020)	0.157 (0.091)
elasticsearch/elasticsearch	**−0.990** (**−0.984**)	0.685 (0.625)	−0.005 (−0.013)	0.182 (0.203)
mrdoob/three.js	**−0.971** (**−1.009**)	0.418 (0.373)	−0.008 (−0.013)	0.414 (0.362)
jekyll/jekyll	**−0.946** (−0.451)	0.057 (0.009)	−0.001 (0.002)	−0.007 (−0.008)
meteor/meteor	**−0.930** (**−0.869**)	0.447 (0.361)	−0.014 (−0.018)	0.340 (0.287)
zurb/foundation	**−0.925** (**−0.926**)	0.145 (0.136)	−0.003 (−0.007)	0.113 (0.116)
joyent/node	**−0.883** (**−0.868**)	0.422 (0.342)	−0.008 (−0.015)	0.315 (0.263)
dogecoin/dogecoin	**−0.861** (**−0.853**)	0.294 (0.234)	−0.014 (−0.021)	0.251 (0.191)
angular/angular.js	**−0.831** (**−0.796**)	0.485 (0.291)	−0.002 (−0.006)	0.352 (0.220)
docker/docker	**−0.802** (**−0.760**)	0.617 (0.493)	−0.002 (−0.004)	0.423 (0.337)
robbyrussell/oh-my-zsh	**−0.796** (**−0.79**)	0.099 (0.072)	−0.004 (−0.009)	0.081 (0.048)
fog/fog	**−0.777** (**−0.761**)	0.575 (0.504)	−0.006 (−0.008)	0.489 (0.426)
jquery/jquery	**−0.737** (**−0.579**)	0.109 (0.041)	−0.011 (−0.012)	0.080 (0.023)
bitcoin/bitcoin	**−0.729** (**−0.713**)	0.277 (0.216)	−0.007 (−0.012)	0.131 (0.109)
bcit-ci/CodeIgniter	**−0.694** (**−0.625**)	0.188 (0.128)	−0.006 (−0.006)	0.089 (0.048)
strongloop/express	−2.114 (**−1.885**)	0.352 (0.188)	**−0.043** (−0.112)	0.363 (0.180)
mbostock/d3	−1.825 (**−1.909**)	0.409 (0.343)	**−0.084** (−0.116)	0.417 (0.337)
antirez/redis	−1.542 (−1.619)	0.367 (0.244)	**−0.031** (**−0.064**)	0.381 (0.245)
ajaxorg/ace	−1.249 (**−1.169**)	0.233 (0.215)	**−0.027** (−0.037)	0.247 (0.195)
diaspora/diaspora	−0.221 (−0.126)	0.003 (−0.003)	0.003 (0.005)	0.040 (0.035)

MM-estimation was used to estimate the coefficients of the regressors. Bold values for α_3 or $\hat{\alpha_3}$ indicate (i) significance at $p < 0.01$ and (ii) that the corresponding model has a larger coefficient of determination r^2. Only the first 15 projects have at least 10,000 commits. Of those, 11 had the *Log-Log* model as a significant and most reasonable fit, and were used as a selection pool for building coordination networks in Section 5.2. Brackets enclose the corresponding quantities after the commits of single-commit developers have been removed from the analysis

A4 Effect of One-Time Contributors

In order to quantify the extent to which our results of team productivity may be influenced by contributors who committed to a project only once, we identified single-commit developers in all of the studied projects. Figure 14 shows the fraction of one-time contributors in all of the studied projects, validating the intuition that they comprise a sizable part of the development team.

In order to ensure that our results about the scaling of productivity are not qualitatively affected by the large fraction of single-commit developers, we have additionally filtered the commit logs of all projects, filtering out the commits of all developers who committed only once. By this study, we focus on the contributions of a core team of that particularly rules out single-commit developers. Using this filtered commit log, we then recomputed all model fits in the paper. In Tables 5 and 6, we report the scaling exponents. We observe no qualitative changes regarding our observation of decreasing returns to scale. We additionally reanalyzed all individual projects, again filtering out all contributions by single-commit

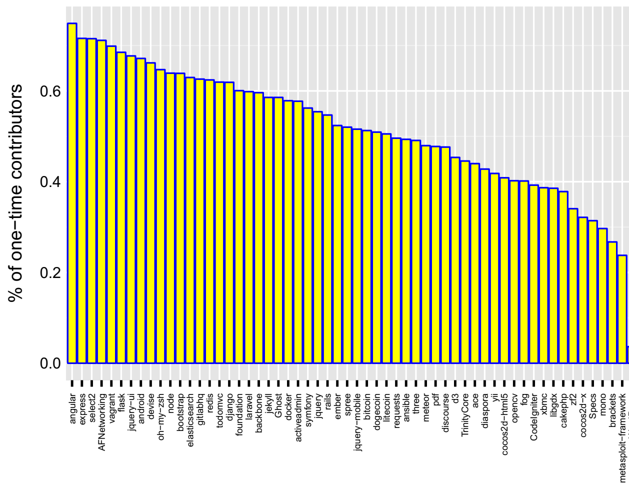

Fig. 14 Fraction of commits submitted by one-time contributors, i.e., developers who never contributed a second commit

Table 5 Estimation of two linear models for Fig. 6 with single-commit developers removed

β_0	α_0	r^2	β_1	α_1	r^2
0.95±0.02	−0.20±0.01	0.10	4.21±0.04	−0.27±0.02	0.04

MM-estimation was used to estimate the coefficients of the regressors. The coefficients are presented together with their corresponding 95 % confidence intervals and are highly significant at $p < 0.001$. The sample size for both models is 13776

Table 6 Estimation of two linear models for Fig. 7 with single-commit developers removed

β_2	α_2	R^2	β_3	α_3	R^2
0.82±0.03	−0.64±0.02	0.32	4.07±0.05	−0.71±0.03	0.22

MM-estimation was used to estimate the coefficients of the regressors. The coefficients are presented together with their corresponding 95 % confidence intervals and are highly significant at $p < 0.001$. The sample size for both models is 13776

developers. We report on the project-wise scaling exponents in the bracketed values in Table 4, again not observing any qualitative changes of our results for individual projects.

A5 Inference Versus Prediction from Linear Models

In Section 4.1 we introduced two linear models[15] as a means of quantifying the negative trends observed in Figs. 6 and 7. In particular we introduced

$$\begin{aligned} \log\langle n\rangle &= \beta_0 + \alpha_0 \cdot \log s + \epsilon_0 \\ \log\langle c\rangle &= \beta_1 + \alpha_1 \cdot \log s + \epsilon_1 \\ \log\langle n'\rangle &= \beta_2 + \alpha_2 \cdot \log s + \epsilon_2 \\ \log\langle c'\rangle &= \beta_3 + \alpha_3 \cdot \log s + \epsilon_3 \end{aligned} \quad (7)$$

where $\langle n\rangle$ is the mean number of commits per active devel $\langle n'\rangle$ is the mean number of commits per team member, $\langle c\rangle$ is the mean commit contribution per active developer, $\langle c'\rangle$ is the mean contribution per team member and $\epsilon_{0,1,2,3}$ denote the errors of the models.

We note that for these models to provide reliable predictions[16] the following conditions must be met: (a) Var($\epsilon_{0,1,2,3,4}|\log s$) = σ^2, for all s (homoskedasticity), (b) $\epsilon_{0,1,2,3} \sim \mathcal{N}$(0, σ_2) (normality of the error distribution) and (c) $E(\epsilon_{0,1,2,3}|\log s) = 0$ (linear model is correct).

We test for homoskedasticity by running the Koenker studentised version of the Breusch-Pagan test (Koenker 1981). This test regresses the squared residuals on the predictor in (7) and uses the more widely applied Lagrange Multiplier (LM) statistics instead of the F-statistics. Although more sophisticated procedures, e.g. Whites test, would account for a non-linear relation between the residuals and the predictor, we find that the Breusch-Pagan test is sufficient to detect heteroskedasticity in our data. The consequence of violating the homoskedasticity assumption is that the estimated variance of the slopes $\alpha_{0,1,2,3}$ will be biased, hence the statistics used to test hypotheses will be invalid. Thus, to account for the presence of heteroskedasticity, we use robust methods to calculate heteroskedasticity-consistent standard errors. More specifically, we use an MMtype robust regression estimator, as described in and provided by the R package *robustbase*.

As for normality of the errors $\epsilon_{0,1,2,3}$, a violation of this assumption would render exact t and F statistics incorrect. However, our use of a robust MM estimator addresses possible non-normality of residuals, as it is resistant to the influence of outliers.

The last assumption pertains to the overall feasibility of the linear model. A common way to assess it is to plot the residuals from estimating (7) versus the fitted values, commonly known as a Tukey-Anscombe plot. A strong trend in the plot is evidence that the relationship between the dependent and independent variable is not captured well by a linear model. As a result, predicting the dependent variable from the calculated slope is likely to be unreliable, especially if the relationship between the variables is highly non-linear.

In Fig. 15 we show the Tukey-Anscombe plots for the four regression models in (7). While we cannot readily observe a prominent trend, we, nevertheless, see two qualitatively different regimes. Specifically the residuals in the lower ranges are close to zero, while they are relatively symmetrically distributed beyond this range. Looking at the line fits in Figs. 6

[15] see (1) and (2).

[16] i.e., to predict the productivity of a team with an arbitrary size given the productivity of a team with a certain size.

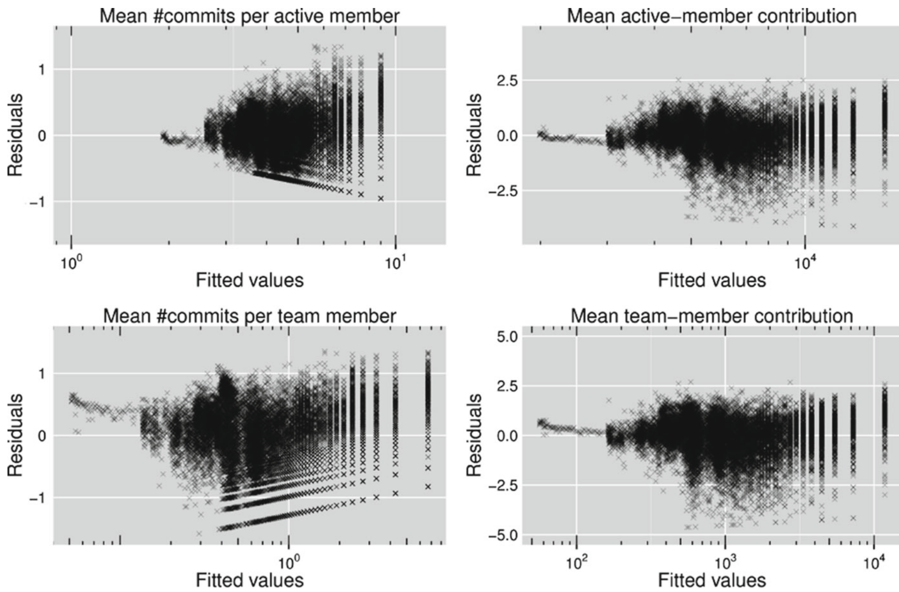

Fig. 15 Residuals versus fitted values for (7). The titles above each plot correspond to the respective scatterplots in Figs. 6 and 7

and 7 we see that the reason for this discrepancy are the outliers in the region of large team sizes, which fall close to the fitted regression lines. Therefore the residuals corresponding to these outliers will be close to zero. Investigating these specific data points reveals that they belong exclusively to the *Specs* project.

To actually quantify a possible trend in Fig. 15 we calculate the normalized mutual information (NMI) between the residuals and the fitted values.[17] As expected the NMI is rather low - 0.04 (top-left), 0.02 (top-right), 0.04 (bottom-left) and 0.03 (bottom-right) - an indication that there is no pronounced systematic error in the linear model. However, even though the NMI values are low, we find that there are all statistically different from zero at p = 0.05.[18]

Therefore, despite the evidence against a systematic error in these linear models, assumption (c) cannot be technically satisfied. We, thus, conservatively avoid using the linear models for predictive means. Since the NMI values, however, are rather low, the regression models are sufficient for our purposes of simply quantifying the observed negative trends. We argue that the *practical* significance of such small effect sizes is negligible with respect to introducing strong systematic errors that could obscure a salient non-linear relationship. Effectively, we can only retroactively infer a significant negative relationship between team size and productivity, but cannot forecast team production given team size. We caution that such inference is also a subject to high variability, as indicated by the low r^2 values (see Section 5.2), and is thus valid only *on average*.

Finally, an argument against the significance of the slopes in (7) is the relatively large sample size of $N = 13998$. Known as the "p-value problem" (Lin et al. 2013), the issue

[17] see Section 5.2 for introduction of the NMI.

[18] statistical significance is judged by the bootstrap approach presented also in Section 5.2

pertains to applying small-sample statistical inference to large samples. Statistical inference is based on the notion that under a null hypothesis a parameter of interest equals a specific value, typically zero, which represents "no effect". In our example, we are interested in estimating the slopes $\alpha_{0,1,2,3}$ with an associated null hypothesis that sets them to zero. It is precisely this representation of the "no effect" by a particular number that becomes problematic with large samples. In large samples the standard error of the estimated parameter becomes so small that even tiny differences between the estimate and the null hypothesis become statistically significant. Hence, unless the estimated parameter is equal to the null hypothesis with an infinite precision, there is always a danger that the statistical significance we find is due to random fluctuations in the data. One way to alleviate the issue is to consider the size of the effect (as we did above) and assess whether the practical significance of the effect is important for the context at hand, even if it is significant in the strict statistical sense.

Another way is to demonstrate that the size and significance of the effect cannot arise by a random fluctuation. To this end we again resort to a bootstrap approach. For each scatterplot in Figs. 6 and 7, we generate 10, 000 bootstrap samples by shuffling the data points. We then estimate the regression models on each bootstrap sample and record the corresponding slope estimate $\hat{\alpha}_{0,1,2,3}$, regardless of its statistical significance.[19] We find that the slopes of the 10, 000 bootstrapped regression models are restricted in the ranges $[-0.02, 0.02]$, $[-0.04, 0.04]$, $[-0.04, 0.03]$ and $[-0.04, 0.06]$ for $\hat{\alpha}_0$, $\hat{\alpha}_1$, $\hat{\alpha}_2$ and $\hat{\alpha}_3$, respectively. Comparing those ranges to the empirical slopes in Tables 1 and 2, we see that by eliminating the relationship between team size and productivity we cannot reproduce the strength of the negative trend found in the dataset. It is the information lost from the shuffling procedure that accounts for the statistical significance of $\alpha_{0,1,2,3}$. Hence, it is safe to conclude that our analysis does not suffer from spuriously significant results introduced by large samples.

A6 Calculating Overlapping Source Code Regions

Our method of identifying overlapping source code changes between co-edits of the same file is based on the information in the chunk header of a *diff* between two versions of a committed file. Such a file diff shows only those portions of the file that were actually modified by a commit. In *git* parlance these portions are known as *chunks*. Each of these chunks is prepended by one line of header information enclosed between @@ . . . @@. The header indicates the lines which were modified by a given commit to this file. Therefore, from all chunk headers within a file diff we can obtain the line ranges affected by the commit and eventually calculate the overlapping source code regions between two different commits to the same file.

As a concrete example, assume a productivity time window of 7 days in which the file *foo.txt* was modified first by developer A and then by developer B in commits C_A and C_B, respectively. The diff of *foo.txt* in commit C_A may contain the following chunk header:

```
@@ -10,15    +10,12 @@
```

The content of the header is split in two parts identified by "−"and "+": −10,15 and +10,12. The two pairs of numbers indicate the line ranges, outside which the two versions (before and after C_A) of foo.txt are identical. More specifically, -10,15 means that starting from line 10, C_A made changes to the following 15 lines, i.e., it affected the line range

[19] $\hat{\alpha}$ refers to the slope of a bootstrap sample, whereas α is the slope of the original regression models.

[10 - 25]. What the result of these changes was is given in the second part of the header. +10,12 indicates that starting from line 10 in the new state of the file, the following 12 lines are different compared to the [10 - 25] line range. Beyond these 12 lines, the old and the new state of *foo.txt* are identical, provided there are no more chunks in the file diff. Therefore, the line range [10 - 25] in the old state of foo.txt and the line range [10 - 22] in the new state after C_A, are the only differences introduced by the commit. This could be caused for example by the removal of three lines from the line range [10 - 25], together with other modifications in the same range.

Since C_A comes prior to C_B in our example, we associate the second part of the chunk header, i.e., line range [10 - 25], to C_A as it represents the state of *foo.txt after* the changes from C_A were applied and *before* those from C_B. Now assume that the diff of *foo.txt* in C_B has only one chunk with the following header:

@@ −10,30 +10,40 @@

In other words, lines [10 - 40] from the old state of *foo.txt* were modified by C_B, and the changes are reflected in lines [10 - 50] in the new state of *foo.txt* after C_B.[20] Note that, lines [10 - 40] represent the state of foo.txt after C_A, but before C_B. Therefore to compute the overlapping source code ranges between C_B and C_A, we need to compare the line ranges [10 - 40] and [10 - 25] and calculate the overlap. In this case, the overlap is 15 lines, which is the weight we attribute to the coordination link from developer *B* to *A* in this particularly simple example. The procedure described above is applied to all pairs of commits by different developers which have edited a common file within a given productivity time window of 7 days. Processing the chunk information in the above way thus allows us to extract linebased, weighted and directed co-editing networks which capture the association between developers and source code regions.

References

Adams P, Capiluppi A, Boldyreff C (2009) Coordination and productivity issues in free software: the role of brooks' law. In: IEEE International Conference on software maintenance. ICSM 2009, pp 319–328. doi:10.1109/ICSM.2009.5306308

Alali A, Kagdi H, Maletic J (2008) What's a typical commit? a characterization of open source software repositories. In: The 16th IEEE International Conference on program comprehension. ICPC 2008, pp 182–191. doi:10.1109/ICPC.2008.24

Albrecht AJ (1979) Measuring application development productivity. In: Proceedings of the joint SHARE, GUIDE and IBM application development symposium, pp 83–92

Arafat O, Riehle D (2009) The commit size distribution of open source software. In: 42nd Hawaii International Conference on system sciences, HICSS'09, pp 1–8. doi:10.1109/HICSS.2009.421

Banker RD, Kauffman RJ (2004) 50th anniversary article: The evolution of research on information systems: a fiftieth-year survey of the literature in management science. Manag Sci 50(3):281–298. doi:10.1287/mnsc.1040.0206

Banker RD, Kemerer CF (1989) Scale economies in new software development. IEEE Trans Softw Eng 15(10):1199–1205

Banker RD, Slaughter SA (1997) A field study of scale economies in software maintenance. Manag Sci 43(12):1709–1725

[20] In particular, C_B added 10 new lines, together with other changes in the range [10-40].

Banker RD, Chang H, Kemerer CF (1994) Evidence on economies of scale in software development. Inf Softw Technol 36(5):275–282
Blackburn JD, Scudder GD, Van Wassenhove LN (1996) Improving speed and productivity of software development: a global survey of software developers. IEEE Trans Softw Eng 22(12):875–885
Blincoe K, Valetto G, Goggins S (2012) Proximity: a measure to quantify the need for developers' coordination. In: Proceedings of the ACM 2012 conference on computer supported cooperative work, CSCW '12. ACM, New York, pp 1351–1360. doi:10.1145/2145204.2145406
Blincoe K, Valetto G, Damian D (2013) Do all task dependencies require coordination? the role of task properties in identifying critical coordination needs in software projects. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, ESEC/FSE 2013. ACM, New York, pp 213–223. doi:10.1145/2491411.2491440
Blincoe KC (2014) Timely and efficient facilitation of coordination of software developers' activities. PhD thesis, Drexel University, Philadelphia, p aAI3613734
Boehm BW (1984) Software engineering economics. IEEE Trans Software Eng 10(1):4–21. doi:10.1109/TSE.1984.5010193
Boehm BW, Clark H, Brown R, Chulani MR, Steece B (2000) Software cost estimation with Cocomo II with Cdrom, 1st edn. Prentice Hall PTR, Upper Saddle River
Brooks FP (1975) The mythical man-month. Addison-Wesley
Cataldo M, Herbsleb J (2013) Coordination breakdowns and their impact on development productivity and software failures. IEEE Trans Softw Eng 39(3):343–360. doi:10.1109/TSE.2012.32
Cataldo M, Wagstrom PA, Herbsleb JD, Carley KM (2006) Identification of coordination requirements: implications for the design of collaboration and awareness tools. In: Proceedings of the 2006 20th anniversary conference on computer supported cooperative work, CSCW '06. ACM, New York, pp 353–362. doi:10.1145/1180875.1180929
Cataldo M, Herbsleb JD, Carley KM (2008) Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: Proceedings of the second ACM-IEEE international symposium on empirical software engineering and measurement, ESEM '08. ACM, New York, pp 2–11. doi:10.1145/1414004.1414008
Chidambaram L, Tung LL (2005) Is out of sight, out of mind? An empirical study of social loafing in technology-supported groups. Inf Syst Res 16(2):149–168. doi:10.1287/isre.1050.0051
Comstock C, Jiang Z, Davies J (2011) Economies and diseconomies of scale in software development. J Softw Maint Evol Res Pract 23(8):533–548. doi:10.1002/smr.526
Dabbish L, Stuart C, Tsay J, Herbsleb J (2012) Social coding in github: Transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 conference on computer supported cooperative work, CSCW '12. ACM, New York, pp 1277–1286. doi:10.1145/2145204.2145396
Earley PC (1989) Social loafing and collectivism: a comparison of the united states and the people's republic of china. Adm Sci Q:565–581
German DM (2006) A study of the contributors of postgresql. In: Proceedings of the 2006 international workshop on Mining software repositories. ACM, pp 163–164
Gousios G, Kalliamvakou E, Spinellis D (2008) Measuring developer contribution from software repository data. In: Proceedings of the 2008 international working conference on mining software repositories, MSR '08. ACM, New York, pp 129–132. doi:10.1145/1370750.1370781
Gousios G, Vasilescu B, Serebrenik A, Zaidman A (2014) Lean ghtorrent: Github data on demand. In: Proceedings of the 11th working conference on mining software repositories. ACM, New York, pp 384–387. doi:10.1145/2597073.2597126 MSR 2014
Harison E, Koski H (2008) Does open innovation foster productivity? Evidence from open source software (oss) firms. Tech. rep., ETLA discussion paper
Hindle A, German DM, Holt R (2008) What do large commits tell us?: a taxonomical study of large commits. In: Proceedings of the 2008 international working conference on mining software repositories, MSR '08. ACM, New York, pp 99–108. doi:10.1145/1370750.1370773
Hofmann P, Riehle D (2009) Estimating commit sizes efficiently. In: Boldyreff C, Crowston K, Lundell B, Wasserman A (eds) Open source ecosystems: diverse communities interacting, IFIP advances in information and communication technology, vol 299. Springer, Berlin, pp 105–115. doi:10.1007/978-3-642-02032-2_11
Ingham AG, Levinger G, Graves J, Peckham V (1974) The ringelmann effect: Studies of group size and group performance. J Exp Soc Psychol 10(4):371–384. doi:10.1016/0022-1031(74)90033-X
Jackson JM, Harkins SG (1985) Equity in effort: an explanation of the social loafing effect. J Pers Soc Psychol 49(5):1199

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories, MSR 2014. ACM, New York, pp 92–101. doi:10.1145/2597073.2597074

Karau SJ, Williams KD (1993) Social loafing: a meta-analytic review and theoretical integration. J Personal Soc Psychol 65(4):681

Karau SJ, Williams KD (1995) Social loafing: research findings, implications, and future directions. Curr Dir Psychol Sci 4(5):134–140

Koenker R (1981) A note on studentizing a test for heteroskedasticity. J Econ 17(1):107–112

Kravitz DA, Martin B (1986) Ringelmann rediscovered: the original article. J Pers Soc Psychol 50(5):936–941

Latane B, Williams K, Harkins S (1979) Many hands make light the work: The causes and consequences of social loafing. J Pers Soc Psychol 37(6):822

Lerner J, Tirole J (2002) Some simple economics of open source. J Ind Econ 50(2):197–234. doi:10.1111/1467-6451.00174

Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions and reversals. In: Soviet physics doklady, vol 10, p 707

Lin M, Lucas H, Shmueli G (2013) Research commentary - too big to fail: large samples and the p-value problem. Inf Syst Res 24(4):906–917

Maxwell K, Van Wassenhove L, Dutta S (1996) Software development productivity of european space, military, and industrial applications. IEEE Trans Softw Eng 22(10):706–718. doi:10.1109/32.544349

Mockus A, Fielding RT, Herbsleb J (2000) A case study of open source software development: The apache server. In: Proceedings of the 22Nd international conference on software engineering. ICSE ’00, ACM, New York, pp 263–272. doi:10.1145/337180.337209

Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: apache and mozilla. ACM Trans Softw Eng Methodol 11(3):309–346. doi:10.1145/567793.567795

Paiva E, Barbosa D, Roberto LimaJ, Albuquerque A (2010) Factors that influence the productivity of software developers in a developer view. In: Sobh T, Elleithy K (eds) Innovations in computing sciences and software engineering. Springer, Netherlands, pp 99–104. doi:10.1007/978-90-481-9112-3_17

Premraj R, Shepperd M, Kitchenham B, Forselius P (2005) An empirical analysis of software productivity over time. In: 11th IEEE International Symposium software metrics, 2005. doi: 10.1109/METRICS.2005.8

Ringelmann M (1913) Recherches sur les moteurs animes: Travail de l’homme. Annales de l’Institut National Agronomique 12(1):1–40

Robles G, Koch S, González-Barahona JM (2004) Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In: 2nd ICSE workshop on remote analysis and measurement of software systems (RAMSS), pp 51–55

Scholtes I, Mavrodiev P, Schweitzer F (2015) From aristotle to ringelmann (dataset). doi: 10.5281/zenodo.14831

Shepperd J (1993) Productivity loss in performance groups - a motivation analysis. Psychol Bull 113(1):67–81. doi:10.1037/0033-2909.113.1.67

Shiue YC, Chiu CM, Chang CC (2010) Exploring and mitigating social loafing in online communities. Comput Hum Behav 26(4):768–777. doi:10.1016/j.chb.2010.01.014. http://www.sciencedirect.com/science/article/pii/S0747563210000166 emerging and Scripted Roles in Computer-supported Collaborative Learning

Sornette D, Maillart T, Ghezzi G (2014) How much is the whole really more than the sum of its parts? 1 + 1 = 2.5: Superlinear productivity in collective group actions. PLoS ONE 9(8):e103,023. doi:10.1371/journal.pone.0103023

Steiner ID (1972) Group process and productivity. Social psychology monographs. Academic

Stigler GJ (1958) The economies of scale. JL Econ 1:54

von Krogh G, Spaeth S, Lakhani KR (2003) Community, joining, and specialization in open source software innovation: a case study. Res Policy 32(7):1217–1241. doi:10.1016/S0048-7333(03)00050-7. open Source Software Development

Wagner J (1995) Studies of individualism-collectivism - effects on cooperation in groups. Acad Manag J 38(1):152–172

Williams K, Karau S (1991) Social loafing and social compensation - the effects of expectations of coworker performance. J Pers Soc Psychol 61(4):570–581. doi:10.1037/0022-3514.61.4.570
Williams K, Harkins S, Latane B (1981) Identifiability as a deterrent to social loafing - 2 cheering experiments. J Personal Soc Psychol 40(2):303–311. doi:10.1037/0022-3514.40.2.303
Wolf T, Schroter A, Damian D, Panjer LD, Nguyen TH (2009) Mining task-based social networks to explore collaboration in software teams. IEEE Software 26(1):58–66. doi:10.1109/MS.2009.16
Yetton P, Bottger P (1983) The relationships among group size, member ability, social decision schemes, and performance. Organ Behav Hum Perform 32(2):145–159. doi:10.1016/0030-5073(83)90144-7

Ingo Scholtes is a postdoctoral researcher at the Chair of Systems Design at ETH Zürich. In 2011, he obtained his doctorate degree from the Systems Software and Distributed Systems group at University of Trier. He was involved in the Large Hadron Collider experiment at CERN, designing and implementing a Peer-to-Peer-based framework for large-scale data distribution which is since used to monitor particle collision data from the ATLAS detector. His current research addresses applications of network science in the analysis of data from socio-technical systems found in the context of software engineering, but also from biology and sociology. Since 2014, he is a Junior-Fellow of the Gesellschaft für Informatik.

Pavlin Mavrodiev is a postdoctoral researcher at the Chair of Systems Design at ETH Zürich where he also obtained his doctorate degree in 2014. He has a background in engineering and computer science. His research broadly covers agent-based modelling of collective decisions in social organizations, both in humans and in non-human social animals. Presently, he has focused his attention to interactions in online environments, such as open source software repositories or online social networks, where large time-stamped datasets are available.

Frank Schweitzer is Professor and Chair of Systems Design (http://www.sg.ethz.ch/) at ETH Zurich. He is also associated member of the Department of Physics at ETH Zurich. His research focuses on applications of complex systems theory to the dynamics of social and economic organizations. Frank Schweitzer is a founding member of the ETH Risk Center (http://www.riskcenter.ethz.ch/) and Editor-in-Chief of "ACS - Advances in Complex Systems" (http://www.worldscinet.com/acs/) and "EPJ Data Science" (http://www.epjdatascience.com/).