Empirical Software Engineering manuscript No.
(will be inserted by the editor)

An Empirical Study on the Teams Structures in Social Coding using GitHub Projects

Mariam El Mezouar · Feng Zhang · Ying Zou

Received: date / Accepted: date

Abstract Social coding enables collaborative software development in virtual and distributed communities. Social coding platforms (e.g., GitHub) provide the pull request feature that allows developers to clone a project, make code changes, and request the project owners to review and integrate the code changes to the main stream of a project. The pull request feature has been widely adopted by a large number of GitHub projects, as it minimizes the risk of exposing the projects to the open communities. The efficiency of the pull requests review process depends both on technical (e.g., the code quality) and social (e.g., the connection of a contributor to the project maintainer) factors. However, it is still unclear which social factors have the most impact on the efficiency of the review process.

To identify the social factors, we study the team structures formed by the developers within the projects that adopt the pull-based development model. We build the pull-based networks, where two developers are linked if one has integrated a pull request submitted by the other. We investigate the 7,850 most popular projects on GitHub that are developed in ten programming languages. We identify the network metrics that have a significant association with the speed of processing the pull requests. Specifically, maintaining a strong core of contributors and denser interactions among the developers is associated with faster response and processing of the pull requests. We further find that more than 90% of the studied projects follow 8 dominant team structures out of 18 possible team structures. In the larger projects, only a set of developers is granted review and integration privileges of the pull requests, reflecting a strict decision making process. The small to medium projects are characterized by a small number of core contributors who maintain repeated interactions, and

M. E. Mezouar and F. Zhang
School of Computing, Queen's University, Kingston, Ontario, Canada
E-mail: {mariam, feng}@cs.queensu.ca

Y. Zou
Department of Electrical and Computer Engineering, Queen's University,
Kingston, Ontario, Canada
E-mail: ying.zou@queensu.ca

are able to process the incoming pull requests more efficiently. The evolution of the team structures of projects over time reveals that only a low percentage of the projects witnesses a change towards team structures associated to faster pull requests processing (e.g., stronger centralization).

1 Introduction

Social coding websites (e.g., GitHub), provide a friendly platform for source code management, issue tracking, and networking among distributed communities (Dabbish et al. 2012). The open source software development benefits from social coding websites, by improving collaboration (Dabbish et al. 2012). The core of many social coding websites is the pull request feature (a.k.a. the pull-based development model) (Barr et al. 2012). A developer (i.e., *contributor*) is free to create a local copy of the project repository, make code changes, and submit a pull request to the project owner. A project owner, maintainer, or integrator is responsible to respond to a pull request by reviewing the code changes and determining if the pull request can be integrated into the main branch of the project.

The pull-based development model eliminates the need for a shared repository, lowers the upfront coordination, and decreases the barriers for the first-time contributors (Gousios et al. 2014). As such, many projects adopt the the pull-based development model, as a substitution to the past collaboration channels, such as submitting patches via issue tracking systems and/or mailing lists (Bird et al. 2007)(Gharehyazie et al. 2015). In terms of popularity, a study by Gousios et al. (2014) reports that pull requests and shared repositories are equally used among GitHub projects ($\approx$ 14% of the projects each), with the remaining projects being single-developer projects. The pull-based development model is particularly appreciated for separating the development effort from the decision making process about the submitted changes (Gousios et al. 2014).

The performance of the pull-based development model depends not only on the quality of the submissions made by the contributors but also on the processing of the pull requests by the project maintainers (particularly, the integrators of pull requests) (Gousios et al. 2015). A qualitative study by Gousios et al. (2016, 2015) shows that the integrators struggle to review or motivate other developers to review the submitted changes. The lack of responsiveness of the project maintainers is a common complaint from the contributors(Gousios et al. 2016). The low responsiveness in processing pull requests delays the integration of code changes on new features and bug fixes, therefore it weakens the power of the pull-based development model.

In this paper, we study the types of team structures that possibly impact the performance of the pull-based development model. We build the pull-based networks of 7,850 GitHub projects. In a pull-based network, two developers are connected if one of them has merged at least one pull request that was submitted by the other. We describe the pull-based networks by a set of network metrics, such as the centralization, and the reciprocity. The network metrics capture the roles of developers as integrators or contributors or both (i.e., reciprocity). The network metrics can also identify the existence of core developers, or equally important participants (i.e., centralization).

We use the network metrics of the pull-based networks to infer the team structures formed in the GitHub projects. A team structure reflects how a development team self-organizes as they submit and review the pull requests. We systematically identify the set of existing team structures based on a set of influential network metrics from the open source projects. Specifically, we investigate the following four research questions:

RQ1. What are the influential network metrics on the performance of the pull-based development model?

We compute the network metrics of the pull-based networks. Then, we compute four performance metrics that reflect the productivity and efficiency of a team in managing the pull requests. We build a regression model to identify the influential network metrics on the performance of a team. We find that three metrics (i.e., density, out-degree centralization and reciprocity) are significantly associated with all four performance metrics.

RQ2. What are the common team structures in the pull-based development model?

We capture the team structures using the three influential network metrics that are identified in RQ1. We define the possible team structures by discretizing the values of the three influential network metrics (e.g., the out-degree centralization metric is discretized into 3 levels based on its distribution). We obtain 18 (i.e., $3\times3\times2$) structures in total. We observe that 8 dominant team structures are adopted by over 90% of the projects. More than a third of the projects follow a team structure characterized by developers taking dedicated roles, and disconnected sub-teams working on different parts of the project.

RQ3. Are there team structures that yield higher performance in processing the pull requests?

We attempt to rank the 8 dominant team structures based on the performance of the associated projects. The team structures describing well-connected teams with a small number of core contributors exhibit the highest performance.

RQ4. Does changing the team structure over time have an impact on the performance of the pull-based development model?

The team structure of projects evolve over time. We compute the team structures and the performance metrics of a project at different temporal snapshots. The adoption of more desirable team structures that we identify in RQ3 is strongly associated to an improvement in the performance of the pull-based development model.

Paper organization. We describe the related work in Section 2. The background on the pull-based network is presented in Section 3. We present the experimental setup of the empirical study in Section 4, followed by our results in Section 5. We discuss the threats to validity in Section 7, and conclude in Section 8.

2 Related Work

In this section, we first present the related work on the governance of open source projects, followed by the evaluation of pull requests. We then discuss the developer social networks and their impact on the development process.

2.1 Governance in Open Source Projects

The governance of software projects is a process, by which the projects are strategically managed, to control the progress and continuous commitment of the developers (Capra et al. 2008). O'Mahony and Ferraro (2007) argue that although the online communities that form the open source projects are enabled by technology, they are not immune to the well-known general principles of organizing. Even if the technical contributions of the developers are a vital part to the progress of the open source projects, O'Mahony and Ferraro (2007) further argue that the process of coordinating the developers became vital to leadership, particularly as projects become mature. As such, a line or work emerged to investigate the social and informal structures of open source projects (Rigby et al. 2013; Crowston and Howison 2006; Dinh-Trong and Bieman 2005). In a study by (Rigby et al. 2013), the relationship between open source project governance and distributed version control is investigated. Similarly to our study, the relationship between two developers x and y is defined with the number of times x signed off or reviewed the code change of y and vice versa. Accordingly, it is found that large open source projects are oligarchies or dictatorships that have a large number of external contributors who do not have the sign-off authority. In an effort to examine the social structure of open source projects, Crowston and Howison (2006) look into the interactions related to the bug fixing process, through the issue tracking systems. The study reveals that although the project teams are highly hierarchical, the centralization levels tend to vary and are negatively correlated to project size, suggesting that large projects are more modular. Dinh-Trong and Bieman (2005) investigate the common characteristics in the development processes of successful open source projects. For instance, the FreeBSD project follows prescribed processes that determine developers' responsibilities, deal with enhancements and defects, and manage releases. Both the FreeBSD and Apache projects have a small set of core developers who control the code base. Bird et al. (2008) study the latent sub-communities from the email social network of several projects to understand how successful open source projects can self-organize. It is revealed that a strong community structure existed within the communication patterns of the participants, and that the structure was more modular when the discussions in the emails focused directly on source code artifacts. Additionally, sub-communities within a project were also representative of the collaboration behavior of the developers. In terms of developers' roles, Joblin et al. (2017) propose a relational perspective to classify developers into core and peripheral using network metrics. The authors further report that core developers exhibit upper positions in the hierarchy, high positional stability, and are at the centre of coordination with other developers.

In prior work, the social structures of open source projects has been captured using metrics such as hierarchy (Rigby et al. 2013), and centralization (Crowston and Howison 2006). Given the rich insights that could be obtained from network structure, we include in our study a more comprehensive set of network metrics (Butts et al. 2008) shown in Table1, and we attempt to identify the most significant metrics in the context of the pull-based development model (RQ1) in order to capture the team structures (RQ2).

2.2 Evaluation of Pull Requests

In social coding, the evaluation of the pull requests made by external contributors plays a key role in the success of distributed software development. The evaluation of external contributions involves both social (Ducheneaut 2005)(Marlow et al. 2013)(von Krogh et al. 2003) and technical factors (Jiang et al. 2013)(Mockus et al. 2002)(Rigby and Storey 2011). Prior work has found that the social and technical impressions of external contributors influence the evaluation of their contributions (Tsay et al. 2014a)(Tsay et al. 2014b). Particularly, Tsay et al. (2014a) found that project integrators are likely to consider both the technical quality of the contribution and the social connection of the contributor to the project integrators. For instance, pull requests with many comments were less likely to be accepted, and their acceptance was dependent on the submitter's prior interaction with the project. Moreover, Tsay et al. (2014a) report that well-established projects were more conservative in accepting pull requests. In addition to the technical factors, such as code quality (Gousios et al. 2015; Tsay et al. 2014a), adherence to project conventions (Gousios et al. 2015), and inclusion of test code in the pull request (Gousios et al. 2015; Tsay et al. 2014a), the study by Gousios et al. (2014) shows that the time it takes to accept and merge a pull request is also influenced by the previous track record of a developer. Yu et al. (2014a) propose to recommend a pull request reviewer based on comment networks of projects, since the review process is mostly embedded in the discussion section of the pull request. Vasilescu et al. (2015) study the effect of introducing continuous integration to the pull request process. The continuous integration is found to be associated to more pull requests being processed to be either accepted and merged or rejected, without compromising the quality of the source code.

Given the importance of the social aspects in the evaluation of the pull requests as shown by previous studies (Gousios et al. 2014)(Tsay et al. 2014a), we complement the existing line of study by investigating the team structures formed within the pull-based development model (RQ2), and their association with the productivity and efficiency of processing the pull requests (RQ3).

2.3 Developer Social Networks

Distributed development is very common for OSS projects. A number of studies (Ehrlich and Cataldo 2012; Wolf et al. 2009; Zanetti et al. 2013) have investigated the social aspects of distributed development, and their relation to the collective and individual performance of a distributed team. Such studies examine the networks formed by the developers as they contribute, communicate, and possibly thrive in their respective communities. There are many ways to build a developer network. Two developers can be connected if they communicated in a discussion thread in the past, thus forming a communication network. Bettenburg and Hassan (2010) and Wolf et al. (2009) report that the structure of communication networks shows an associated to future failures, in addition to the quality of bug reports, as revealed by Zanetti et al. (2013). In the communication network of a large software project, Ehrlich and Cataldo (2012) found that the centrality of developers in the network indicates their

performance in fixing bugs. The *follow* networks of developers capture the follow behaviours among developers in social coding platforms, such as GitHub. Schall (2014) examine the *follow* network of developers on GitHub to recommend who to follow. The purpose is to help developers build a reputation and a strong network among their peers (Schall 2014). Yu et al. (2014b) mine the *follow* networks, and identify the behaviour patterns of developers from the networks (e.g., star, group, or hub shaped). Yu et al. (2014b) further claim that the identified behavior patterns can inform the design of assistive tools for developers, such as recommendation systems.

Pull-based collaboration networks capture a different layer of interactions among developers in social coding platforms. In the context of the pull-based development, the collaboration happens when one developer reviews the pull request made by others, as studied by Rigby et al. (2013). However, the hierarchy of the networks is the only aspect studied by Rigby et al. (2013) in their investigation of the review networks of developers. In our paper, we perform a more comprehensive study and include other network metrics to capture the team structures of the projects in the pull-based development model. Additionally, we also look into the evolution of the pull-based networks over time (RQ4).

3 Pull-based Networks

In this section, we provide a background of the pull-based software development, describe the pull-based networks inferred from the existing open source projects, and discuss the metrics used to capture the performance of the pull-based development model.

3.1 Pull-based Software Development

The pull-based development model has become the de facto standard of collaboration within open source projects (Gousios et al. 2015). There are two types of roles for developers to participate in a pull-based model: 1) *contributors* who make the code changes and submit the pull requests; and 2) *integrators* who are responsible to review pull requests and decide whether to merge the pull requests to the main code base. A contributor can either be part of the project maintainers or an external developer to the project. An integrator is, on the other hand, necessarily part of the team that maintains the project. In some projects, the project maintainers can directly commit their changes to the code base; while external developers need to create pull requests to submit their changes. In other projects, the project maintainers and external developers can both solely use pull requests to submit code changes. In this case, pull requests are used to track, review, and discuss all the code changes (Gousios et al. 2015).

On social coding websites, such as GitHub, and BitBucket, the contextual and structured information are recorded for each pull request. For instance, a single pull request contains three tab pages on GitHub as shown in Figure 1: 1) the "Conversation" tab page is used to track the discussions and activities related to the pull request; 2) the "Commit" tab page shows all the commits associated with the pull request; and 3)

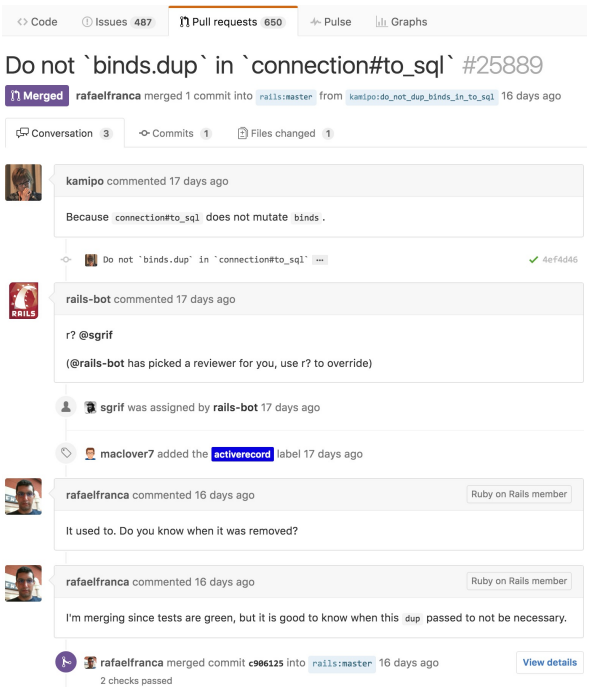


Fig. 1: Example of a pull request

the "Files changes" tab page lists all files changed in the pull request and records the differences resulting from each code change.

3.2 Pull-based Network

We define a pull-based network as a directed and weighted graph. Each node represents a developer. An edge between two nodes signifies that two developers have engaged in a <contributor, integrator> collaboration. The edges of the network are weighted by the number of times that two developers collaborated in the past. We conjecture that the <contributor, integrator> relationship constitutes collaboration between two developers, because the review process of a pull request involves both a review of the code submitted, along with back-and-forth discussions to request changes if needed. The network includes all the developers who have either submitted a pull request, reviewed and integrated a pull request, or both (regardless of the developers' level of participation in the project). For each project, we build a pull-based network, as shown in Fig. 2. We represent the pull-based networks as a set of vectors with each vector in the form of *Contributor, Integrator, Number_collaborations*. We show in Table 1 a descriptive summary of the pull-based networks of the 7,850 GitHub projects.

It is possible to mirror the network structure of development teams using other types of relationships, such as the co-editing of files or the participation in commu-

Table 1: A descriptive summary of the pull-based networks extracted from the GitHub projects

Number of developers	less than 5	between 5 and 10	between 11 and 20	between 21 and 50	between 51 and 100	more than 101
# projects	1918	1848	1717	1578	525	264
Percent (%)	24.43	23.54	21.87	20.10	6.69	3.36
Avg # contributors	2.53	6.38	12.79	29.25	65.34	218.49
Avg # integrators	1.91	3.16	4.29	4.98	6.56	11.06
Avg # of developers acting as both contributors and integrators	0.97	1.79	2.47	2.76	3.98	8.04
Avg # connections	3.35	8.92	17.96	37.19	83.31	299.16
Avg # bi-directional connections	0.58	1.03	1.50	1.58	2.41	5.42
Avg weight of connections	5.56	3.98	3.15	2.50	2.23	2.35

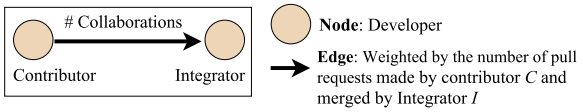


Fig. 2: Construction of the pull-based collaboration network

nication threads. However, in this paper, we purposefully investigate the high level structure of the development teams, as reflected by the review process of the pull-based development model (i.e., the <contributor, integrator> relationship). Our goal is to infer from the constructed networks the structures of the development teams in the pull-based development model.

We use the network metrics to describe the pull-based networks. From the network metrics, we can infer information such as the centralization of the developers, or how densely the developers in a network are connected. Specifically, network metrics are used to describe the structural properties of a network in its entirety (Anderson et al. 1999), in terms of centralization (Freeman 1977, 1978), informal organization (Krackhardt 1994), and general structure (Garlaschelli and Loffredo 2004). We compute 10 commonly used network metrics to understand the team structures in the context of the pull-based development model. We capture the team structures based on a discretization of the most influential network metrics that we identify in RQ1. As such, we are able to systematically define the different structures formed by the developers as they collaborate through the pull-based model. Table 2 shows the list of the network metrics and the corresponding descriptions.

3.3 Performance Metrics for Evaluating the Pull-based Model

It is important to process the incoming pull requests in an efficient and productive manner in order to maximize the benefits of the pull-based model. In previous studies on the pull-based development model (Gousios et al. 2014; Yu et al. 2015), models

are built to predict the decision to merge a pull request, and the time it takes to process it.

In our study, we focus on the responsiveness of the team in processing the pull requests. Since it is not our goal to identify the factors behind a pull request acceptance, we do not consider the decision to merge a pull request as an outcome metric, but we include the time to process a pull request. Moreover, we add three metrics, i.e., the ratio of long running pull requests, the number of pull requests closed daily, and the response time. We explain the performance metrics in more details below.

3.3.1 Productivity

We compute the following two metrics to capture the productivity of a development team. The productivity metrics are designed to assess whether the developers are able to produce the intended results, i.e. closing the pull requests, within a time period.

- *The ratio of long running pull requests.* GitHub defines a long running pull request as one that has lived for more than a month, with some activity (e.g., a comment) within the past month (Rick 2013). This metric helps us assess whether the team leaves pull requests lingering for an extended period of time. The higher the ratio of the long running pull requests, the lower the productivity of the team.
- *The average number of pull requests closed daily.* The higher the average, the more productive the team is. As more pull requests are closed, more issues are fixed and more new features are introduced to the project.

3.3.2 Efficiency

We extract the following two metrics to quantify the efficiency of a development team. The efficiency metrics are meant to measure whether the developers process the pull requests using the least amount of resources, i.e., time.

- *The average response time*. The time it takes project maintainers to provide a first response to the pull request. The sooner project maintainers provide an initial feedback to the contributor, the more likely the contributor is motivated to work on the requested reviews to improve the quality of the code change.
- *The average processing time*. The time it takes the team to process and close a pull request. The lower the processing time, the sooner the integrators can focus on processing other pull requests, and the sooner contributors can work on new code changes.

To ensure that the performance metrics can capture distinct information, we compute the pairwise correlation among the collected metrics using the Spearman's rank coefficient. We choose Spearman's rank correlation test over other non-rank correlation tests (e.g., Pearson's coefficient) because rank correlation is more robust to data that is not normally distributed (Zar 2005). For each pair of metrics, we find that the value of the Spearman's rank coefficient is always less than 0.7 (i.e., the recommended threshold by Zar (2005)). Therefore, we use all four metrics to measure the productivity and the efficiency of a development team.

Table 2: The extracted network metrics

GLI Category	GLI	Description	Purpose
Global structure (Garlaschelli and Loffredo 2004)	Density	The ratio of the number of edges to the number of possible edges.	Measures the sparsity of the connections in a graph.
	Reciprocity	The fraction of edges which are symmetric (reciprocal edges). It can include the null edges (i.e., `reciprocity_1`) or only the mutual edges (i.e., `reciprocity_2`).	Measures the likelihood of nodes in a directed graph to be mutually linked.
	Transitivity	The fraction of triangles in a graph relative to the total number of connected triples of nodes in the graph. It is computed in two ways: strong transitivity (i.e., `transitivity_1`): $(i,j),(j,k) \in E \Rightarrow (i,k) \in E, for (i,j,k) \in V$); and weak transitivity (i.e., `transitivity_2`): $(i,j),(j,k) \in E) \Rightarrow (i,k) \in E)$ (where E is the set of graph edges and V is the set of graph vertices).	Measure the tendency of the nodes to cluster together. High transitivity means that the network contains communities or groups of nodes that are densely connected internally
Centralization (Freeman 1977)(Freeman 1978)	In-degree centralization	The metric is computed at the graph level as the total deviation from the maximum observed in-degree centrality score	Measures the presence of central nodes based on incoming edges.
	Out-degree centralization	The metric is computed at the graph level as the total deviation from the maximum observed out-degree centrality score	Measures the presence of central nodes based on outgoing edges.
Informal organization (Krackhardt 1994)	Connectedness	The fraction of all nodes pairs which are not strongly disconnected (i.e., there exists a path that connects the two nodes)	Describe the extent to which the structure of a graph approaches that of a tree.
	Efficiency	Essentially, the degree to which the graph uses as few links as possible to connect the nodes which are already connected in the graph. An index of the number of extra lines in the graph.	
	Hierarchy	The fraction of nodes pairs in the graph which are neither strongly connected nor strongly disconnected, i.e., one node can reach the other through some path, but the other node cannot reach it.	

4 Experimental Setup

In this section, we provide details on collecting and processing the GitHub data. Fig. 3 depicts our experimental setup including the overall approach.

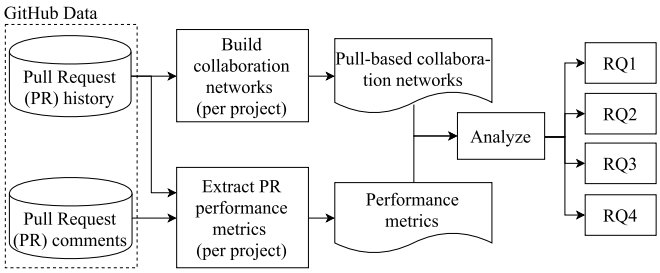


Fig. 3: Overall approach to study the team structures formed within the pull-based networks, and their performances

4.1 Collecting the GitHub Data

GitHub[1] is not only the largest code host (over 38 million repositories), but also a very popular social coding platform. GitHub provides issue tracking, pull requests, commits history, subscriptions to other users, and documentation. Developers can easily share their profile and their activities through GitHub.

To collect the GitHub data, we use GhTorrent (Gousios 2013), an off-line mirror of the GitHub data. GhTorrent has been collecting data since February 2012 and is updated periodically, i.e., every two to three weeks. We download eight temporal snapshots (i.e., 2014-01-02, 2014-08-18, 2015-01-04, 2015-08-07, 2016-02-16, 2016-03-01, 2016-06-01, and 2016-11-01) of the GitHub data dump. We intentionally keep approximately a 6-month interval between each two snapshots whenever possible. The multiple snapshots enable us to study the evolution of the pull-based networks. We apply the following three filters to select the subset of subject projects:

F1. Programming language filter. We choose the projects that are written in the ten most popular programming languages on GitHub: JavaScript, Java, Python, CSS, Php, Ruby, C++, C, Shell, and C#.

F2. Type of project filter. We only extract the non-forked projects. A non-forked project is an original repository that was started from scratch, as opposite to forked projects which are copies of other repositories. At this step, we obtain over six million projects.

F3. Activity level filter. An almost equal number of projects use pull requests and shared repositories for distributed collaboration ($\sim$ 14%) (Gousios et al. 2014). The remaining projects that do not use either collaboration approches (over 60%) are single-developer projects (Gousios et al. 2014). We focus on the most active projects in terms of the number of recorded pull requests, as we need to build pull-based networks. We select the projects in the top 95% percentile, with over 100 recorded pull requests. In total, we obtain 7,850 projects with a total of 2,854,917 pull requests.

[1] https://github.com/

4.2 Computing and Normalizing the Network Metrics

We process the pull-based networks using the *R* package *SNA* (*S*ocial *N*etwork *A*nalysis) developed by Butts (Butts et al. 2008). The *SNA* packages transforms each network into a matrix, and provides a set of functions (e.g., `grecip()`) to compute the network metrics listed in Table 2. Moreover, it is important to include other project measures that have shown to have strong predictive power in the previous studies (Moser et al. 2008; Nagappan and Ball 2007). Therefore, we include the **number of commits** and the **number of developers** overtime, to control the impact of the activity level of a project and the size of the project team. To control the impact of the number of developers, we use a normalized metric $\frac{nodes}{edges}$, where the number of nodes is a simple count of the developers in a project, and the number of edges describes the sparsity of collaborations among the developers.

Reason for the normalization. When two networks have different sizes, it is not recommended to directly compare the values of their associated network metrics (Anderson et al. 1999)(Butts et al. 2008)(de Reus and van den Heuvel 2013). For instance, we assume that two networks N_1 and N_2 have the same centralization value C, but different sizes ($N_1 > N_2$). The centralization of a network measures the importance of the different nodes based on the number of edges. As a network grows in size, its centralization value inevitably changes as well. A centralization value equal to C is within the norm for N_1, compared to other networks of the same size. However, the same centralization C is larger than what is usual for the smaller network N_2. A prior study (Anderson et al. 1999) has shown that the interaction between network metrics and the size of a network can not be ignored. Considering the intrinsic dependence on the size of a network, it is likely that the difference in the metric values can be partly explained by the difference in the network sizes. Therefore, it is important to normalize the network metrics by controlling the effect of the network size. The normalized metrics allow for a more sound interpretation of the network metric values, and a fair comparison of graphs with different sizes (de Reus and van den Heuvel 2013).

The CUG test for normalization. To control the effect of size, we perform the Conditional Uniform Graph (CUG) hypothesis test (Anderson et al. 1999), a simple model that fixes certain properties of a network (e.g., the number of nodes) at particular values, and treats all networks meeting the selected properties as equally probable. The CUG test is adequate for the task of controlling the effect of size on the remaining network metrics, as the effect of size is the only substantial effect reported by the literature (Anderson et al. 1999; Butts et al. 2008; de Reus and van den Heuvel 2013). In the CUG test, a baseline model is built and used as the null hypothesis. Under the baseline model, a number of networks of the same size are used as the input network and are simulated using Monte Carlo simulation (Handcock et al. 2008). Monte Carlo simulation shuffles edges while fixing the number of nodes to simulate the networks for the baseline model. The test generates the distribution of a network metric under the baseline model, and compares the observed network metric to the baseline distribution. To perform the CUG tests, we use *Statnet*, an *R* package developed by Handcock et al. (2008). For each network metric value, the CUG test returns the probability of the observed value to be *greater than or equal to* the values under the baseline model (i.e., $Prob_{greater} = Prob(X <= Observed)$), and the

probability of the observed value to be *less than or equal to* the values under the baseline mode (i.e., $Prob_{less} = Prob(X >= Observed)$).

Normalizing the metrics. To normalize the values of the network metrics, we choose to transform each metric value into $Prob_{greater}$, as we find it easier to interpret. When $Prob_{greater}$ is closer to 1, the value of the network metric is unusually high for networks of the same size. The closer $Prob_{greater}$ is to 0, the smaller is the observed value of the network metric compared to the baseline. For instance, assuming $Prob_{greater} = 0.9$ for the metric centralization in a network, we can conclude that the network is particularly centralized compared to other networks of the same size. Thus, the normalized metric values help us compare the strength of a network property across networks with different sizes.

5 Results

In this section, we present the results of our experiments with respect to four research questions.

RQ1. What are the influential network metrics on the performance of the pull-based development model?

Motivation. In distributed software development, the social and organizational aspects have an impact on the individual and collective performance of the developers (Ehrlich and Cataldo 2012). As such, the performance of the pull-based development model is governed by both technical factors (e.g., the quality of the code changes), and social factors (e.g., the team structure). However, it is unclear which team structure properties have the highest impact on the performance of processing the pull requests. In this research question, we identify the network metrics (described in Section 3.2) that have a significant association with the performance metrics of the pull-based development model (listed in Section 3.3).

Approach. For each subject project, we first build a pull-based network. Second, we compute and normalize the network metrics to describe the structural properties of the pull-based network (see Section 3.2). Finally, we conduct the following steps to identify the influential network metrics.

Reduce highly-correlated metrics
In the presence of highly-correlated metrics, the estimate of the impact of one metric on the dependent variable tends to be less precise, thus weakening the classification model. Therefore, we use the R function `cor()` to generate the correlation matrix of the number of vertices and edges in the network, in addition to the ten network metrics. If the correlation between two metrics is more than 0.7 (i.e., the recommended threshold by Zar (2005)), we select the one which is easier to interpret in the context of the pull-based development model.

Build a regression model

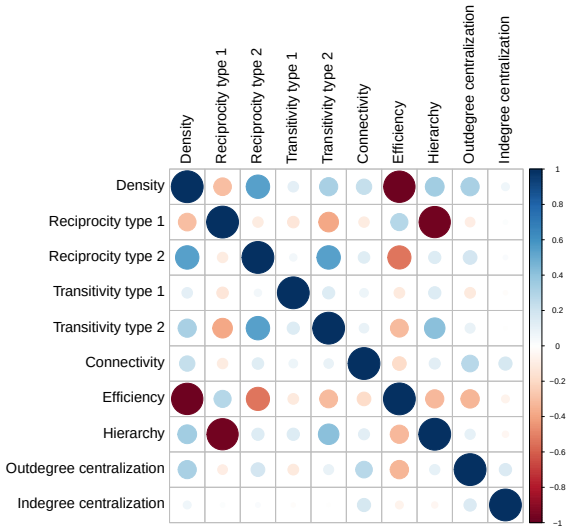


Fig. 4: Correlation analysis of the network metrics

The purpose of the analysis is to model the relationship between the response variable (i.e., the performance metrics, such as the average response time) and the predictors (i.e., the network metrics, such as the density). Therefore, we use linear regression to determine which predictors are statistically significant and how changes in the predictors relate to changes in the response variable.

For each performance metric, we build a separate regression model and use the R^2 metric to assess the fit of the model. The R^2 measures the "variability explained" of the response variable that is analyzed (Steel and Torrie 1960). For instance, an R^2 of 0.5 indicates that 50% of the variability of the response variable is being modeled (i.e., "explained") by the predictors. The remaining 50% of the variability may be due to external factors that are not being modeled or cannot be controlled. The interpretation of R^2 values depends on the analysis that is being performed. For example, when the main goal is prediction, the R^2 values should be very high (e.g., around 0.7 to 0.9) (Choi and Varian 2012). Low R^2 values (e.g., around 20%) may also generate interesting insights in fields such as social sciences or psychology (Bersani et al. 2016).

Identify the influential network metrics

We identify the predictors (i.e., network metrics) that show the highest association with the response variables (performance metrics). The influential predictors can then be used to define the team structures. Therefore, we identify the significant predictors (p-value < 0.05). We also report the regression coefficients of the predictors, to assess the influence of each predictor on the response variable. The regression coefficient tells us how much the response variable (e.g., the response time) is expected to increase when the predictor variable (e.g., the density) increases by one, holding all the other predictors constant. The regression coefficients of different predictors are not

Table 3: Regression coefficients of the significant metrics from the linear regression models

	Regression coefficients			
	Long running pull requests $R^2 = 0.25$	**Pull requests closed daily** $R^2 = 0.28$	**Response time** $R^2 = 0.32$	**Processing time** $R^2 = 0.18$
vertices_over_edges	-31.85	-	-87228.75	-
commits	0.0036	0.0041	-	16983.65
reciprocity_2	-64.99	18.62	-	-456455.56
outdegree_centralization	-0.7126	5.9078	-	-78974.6
density	-		-34455.69	-12465.85

always comparable because the predictors have different types of unit. For example, the response time is measured in seconds, while the number of pull requests closed daily is counted as units of pull requests.

Results. The correlation analysis leads to the removal of two network metrics (i.e., `hierarchy`, and `efficiency`). We show in Fig. 4 the results of the correlation analysis. We retain the metric density (over the efficiency) because it reflects whether developers collaborate with the entire team or only a subgroup of developers in the team. The metric efficiency measures whether the network uses as few edges as possible to connect the developers (see Table 2). Low network efficiency means two developers are indirectly connected more than once, which is not as easy to interpret in this context as the density. Finally, we choose the reciprocity (over the hierarchy) because the reciprocity measures whether the developers take single or multiple roles in the team (i.e., contributors and integrators). The notion of `hierarchy` is not applicable in the pull-based development model because a directed edge from a contributor to an integrator does not indicate hierarchy levels, but rather collaboration.

The four most influential network metrics in terms of their association with the performance of pull-based development model include: the vertices over edges, reciprocity type 2, out-degree centralization, and the density. To select the top influential network metrics, we identify the metrics that return a $p-value < 0.05$. We further report the regression coefficients of the selected metrics, to measure the influence of each predictor. Table 3 shows the regression coeffients of the significant predictors, for each of the linear regressions models (a model is built for each performance metric defined in Section 3.3). We also show in Table 3 the coefficient of determination R^2 of the trained models. The resulting R^2 values are low (i.e., 0.32 or less), therefore, the network metrics can only explain up to 32% of the variability of the performance metrics. Therefore, the team structure properties (as measured by the network metrics) can only partly explain the productivity and efficiency of the development team in processing the pull requests. The remaining variability is likely due to other factors, such as the complexity of the code change in the pull requests and the time availability of developers. However, we can still infer interesting insights about the relationship between the network metrics and the performance metrics. For instance, an increase in one unit of the network density is associated to the decrease

by 12465.85 seconds (3.46 hours) in the processing time of the pull requests. A more dense network implies a developers would collaborate at the pull request level with diverse developers, instead a reduced number of developers. In other words, encouraging more collaboration links among the developers who process the pull requests could be associated to reducing the processing time of the pull requests. Unsurprisingly, an increase in the processing time can occur when the number of commits a projects receives is higher. Every additional commit is associated with an increase of 16983.65 seconds (4.72 hours) in the processing time. A higher reciprocity is possibly associated to lower processing time. A reciprocal link between two developers indicates prior connection between the two developers. Therefore, this result confirms a previous finding by Tsay et al. (2014a) regarding the role that a contributor's prior connection to the project integrators has on the processing of the pull requests.

We find an association between faster response and processing times of the pull requests, and developers taking multiple roles and collaborating densely. The presence of central developers that act as both contributors and integrators is associated with closing more pull requests.

RQ2. What are the common team structures in the pull-based development model?

Motivation. In RQ1, we identify the network metrics that have the highest influence on the performance of processing pull requests. In this research question, we attempt to capture the different structures formed by the developers as they submit or review pull requests, within a large set of GitHub projects (7,850 projects). We use the term team structure to describe the self-organization of developers within the pull-based development model using the contributor and integrator relationship. From the pull-based networks of the GitHub projects, we infer the existing frequently adopted team structures.

Approach.

First, we capture the team structures within the pull-based development model from the selected GitHub projects. We then investigate the frequency of each team structure among the selected GitHub projects, and describe the most frequent team structures found in the selected pool of projects.

To identify the existing team structures adopted by the studied projects, we characterize the pull-based network associated to each project using the influential network metrics identified in RQ1. We present below the interpretation of each network metric in the context of the pull-based development model.

a) Out-degree centralization: measures the importance of contributors based on the activity levels (i.e., the number of submitted pull requests). High out-degree centralization indicates the existence of core contributors; while low out-degree centralization shows that the contributors participate equally.

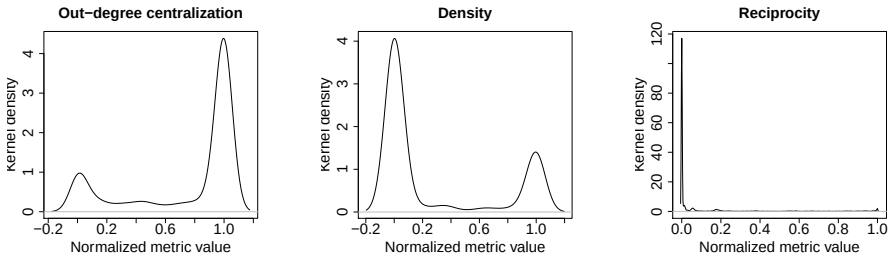


Fig. 5: Distribution of the influential network metrics

b) Density: measures how connected the network of developers is. High density reflects a strongly connected team where the developers have prior interactions with many of their teammates. Low density characterizes teams with sparse connections.

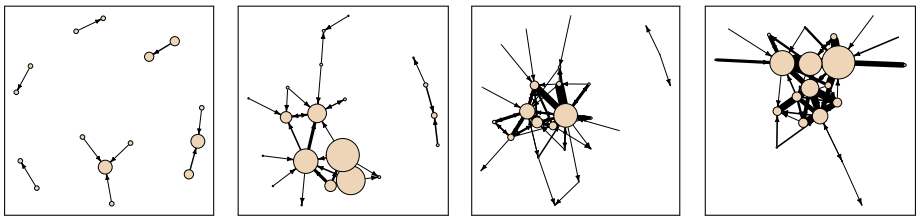
c) Reciprocity: measures the likelihood of developers to both contribute and integrate pull requests. High reciprocity indicates that developers are more likely to take both roles (i.e., integrator and contributor). Low reciprocity shows that developers are likely to have dedicated roles.

Each network is assigned a discrete representation that reflects the team structure adopted by the project, by discretizing the values of the influential network metrics. The discretization is performed in two steps: 1) we normalize the values of the network metrics to the range 0-1 (as explained in Section 4.2), and 2) we perform the transformation into an n-level scale depending on the distribution of each metric. We examine the distributions of the three selected metrics to identify the proper discretization scale. Fig. 5 shows the distributions of the network metrics. We observe that the out-degree centralization and the density both roughly follow a multimodal distribution; while the reciprocity follows an exponential distribution.

We discretize the two metrics **Out-degree centralization** and **Density** using a 3-level scale that captures the two local maxima and the flat area between them. The **Reciprocity** is discretized using a 2-level scale to mirror the initial peak and the flat area that follows. We compute the Spearman correlation among the discretized network metrics, and the metrics that measure the size of projects (i.e., the # of developers, the # of commits, the # of LOC), and we report the coefficients of the pairs that show significant correlation (i.e., $p - value < 0.05$). With the discretized network metrics, we generate 18 (3 x 3 x 2) possible team structures. Each network is assigned a team structure encoded in the form $\text{O}_{\text{utdegree}}$-$\text{D}_{\text{ensity}}$-$\text{R}_{\text{eciprocity}}$. The team structure reflects the strength of each metric (e.g., reciprocity) in the network. We show in Fig. 6 illustrating examples of pull-based networks and their assigned team structures.

Results.

There is a negative moderate correlation between the network metrics and the size of the development team. As a team grows in size, it tends to be less centralized, less dense, and less reciprocal. Prior work has shown that centralization scores are negatively correlated with the number of developers who contributed to



(a) 1-1-1 **(b)** 3-1-1 **(c)** 3-2-2 **(d)** 3-3-1

Fig. 6: Illustrating example of project pull-based networks and their assigned team structures in the form Outdegree-Density-Reciprocity. The size of the node reflects the importance of the developers. An egde goes from a contributor to an integrator.

Table 4: Spearman's correlation coefficients between the network metrics and the activity metrics of the projects

	# of developers	# of commits	# SLOC
Centralization	-0.24	0.29	0.19
Density	-0.56	0.07	0.09
Reciprocity	-0.47	0.04	-

Table 5: The most frequent team structures shown in the form Outdegree-Density-Reciprocity.

Group	Group frequency	Team structure	Freq.	# of Projects	Median # of developers	Median # of commits	Median # SLOC
X-1-2	29.2%	3-1-2	21.4%	1680	28	1508	19433
		2-1-2	4.4%	345	21	706	13702.5
		1-1-2	3.7%	290	36	842	4839
X-1-1	39.9%	3-1-1	19.1%	1499	26	961	15439.5
		1-1-1	14.3%	1122	35	459	5288
		2-1-1	6.5%	510	28	732	9139
3-3-X	24.3%	3-3-2	18.7%	1468	9	1205.5	19757
		3-3-1	5.6%	440	8	784.5	22056

the bug reports (Crowston and Howison 2005; Howison et al. 2006). In the context of contributions to pull requests, our analysis result in similar conclusions related to centralization ($r = -0.24$ and $p < 0.05$). A possible interpretation of this finding is that in a large project, it might not be possible for a single developer to be involved in processing every pull request. As projects grow, they tend to become more modular, with different developers responsible for different modules. A similar finding is observed for the metric *density* ($r = -0.56$ and $p < 0.05$). This further suggests that as a project grows, it becomes challenging for a given developer to maintain collaboration with everyone in the team, and would build relationships with a reduced set only resulting in a less dense network. To conclude, the analysis of the pull-based networks confirms some of the findings resulting from other types of developer networks that are built based on different communication venues (e.g., issue tracking systems).

Out of the 18 possible team structures, 16 structures exist in our selected set of projects with varying frequencies. Half of the existing team structures (i.e., 8) cover more than 90% of the studied projects. We focus on the 8 most frequent team structures that account for the majority of the projects, in order to study a reduced set of team structures. Table 5 shows the 8 frequent team structures, their associated frequencies, and the number of projects. The 8 most frequent team structures can be grouped within three groups. We describe the three groups in the form $O_{utdegree}$-D_{ensity}-$R_{eciprocity}$, where a metric is assigned an X if it varies within a group.

1) Sparse team with multi-role developers (X-1-2): This group includes 3 of the team structures shown in Table 5, with a total frequency of **29.2%**. In a sparse team (i.e., very loosely connected), developers collaborate with a subset of the team only. The developers are likely to act as both contributors and integrators. Within this group, the most frequent team structure (i.e., 3-1-2) describes a development team with few core contributors, who submit most of the pull requests. As shown in Table 5, the team structure 3-1-2 is mainly associated to the smaller projects in terms of the number of developers ($median = 14.64$), and the size of the source code ($median = 31552.76$). It is expected for the smaller open source projects to have a centralized structure, with developers participating as both contributors and integrators of the pull requests. The connections in the pull-based network of this type of team structure are sparse, as would be expected for a smaller or newer open source project. In the remaining and less frequent team structures, we observe varying levels of out-degree centralization ($O_{utdegree}$[1-2]-D_{ensity}[1]-$R_{eciprocity}$[2]), indicating development teams that have different ratios of core contributors.

2) Sparse team with single-role developers (X-1-1): This group is the most frequent (i.e., **39.9%**) and covers 3 of the team structures shown in Table 5. It is similar to the first group as it also describes sparse teams. However, the developers are more likely to take on a single role only, for example as integrators. Within this group, the most frequent team structure (i.e., 3-1-1) describes a sparse development team with few core contributors who submit most of the pull requests, and with mostly single-role developers. The second most frequent team structure in this group (i.e., 1-1-1) highlights sparse development teams with contributors who contribute equally, and take on single roles. Both team structures (i.e., 3-1-1 and 1-1-1) are associated to larger projects in terms of team size (see Table 5). As reflected by the reciprocity metric, the fact that developers take dedicated roles reflects the decision making process in large projects. By assigning developers to specific roles (e.g., deciding or not to integrate the pull requests), the development team is more strict in its structure, in order to maintain the code quality. This observation is conformant with prior work on the characteristics of open source projects (Gacek and Arief 2004)(Rigby et al. 2013), which speculates that a set developers has more power than other developers in making executive decisions. Within this group of team structures, the centralization degree varies from projects with a more centralized power structure (similar to Linux), to more decentralized organizations, as was similarly reported by Gacek and Arief (2004).

3) Well-connected team with core contributors (3-3-X): This group covers 2 of the team structures shown in Table 5, with a total frequency of **24.3%**. In a well

connected team, the developers tend to collaborate with many team members. The team is very centralized around core contributors, who are responsible of submitting most pull requests. The most frequent team structure within the group (i.e., 3-3-1) describes well-connected development teams with core contributors and single role developers. A study on the email communications between developers in the Apache project reveals that a set of core developers self organize into sub-groups that communicate intensely in completing the project (Robertsa et al. 2006). This finding describes a team structure similar to the structure 3-3-X. Additionally, we find that the Apache Ignite project hosted on GitHub[2] also follows the team structure 3-3-2. Therefore, the analysis performed in prior work on other types of developer networks generates similar findings as the pull-based networks.

We capture three groups of team structures from the GitHub projects: sparse teams with multi-role developers, sparse teams with single-role developers, and well-connected teams with core contributors.

RQ3. Are there team structures that yield higher performance in processing the pull requests?

Motivation. The 8 frequent team structures are found in over 90% of the studied projects. It is unclear if all of them are associated to differing performance of the pull-based development model. In this question, we rank the team structures in terms of the productivity and efficiency of the associated projects. Identifying the efficient and productive team structures can provide useful insights to practitioners to improve their current approach to processing the incoming pull requests.

Approach. We investigate the performance of the 8 most frequent team structures. We first identify the projects associated to each team structure. Then, we assign to every team structure the values of the performance metrics (e.g., average response time) of the associated projects. For each performance metric, we rank the team structures using the associated distributions of the performance metric.

We further examine the significance of difference among the team structures. We first perform a Kruskal-Wallis test (Kruskal and Wallis 1952) on the distributions of the metric values of all the team structures. The Kruskal-Wallis test is a non-parametric statistical test to evaluate whether two or more distributions have equally large values. The advantage of using non-parametric statistical methods is that they make no assumptions about the distribution of the data. If the distributions are statistically different (p-value < 0.05), we conclude that at least one team structure is different from the others in terms of the tested performance metric. We determine which team structures are different by performing a multiple comparison test with pairwise comparisons, and by adjusting the p-values for multiple comparisons. We specifically use the R function KRUSKALMC (Siegel 1956) from the package PGIRMESS. The multiple

[2] https://github.com/apache/ignite

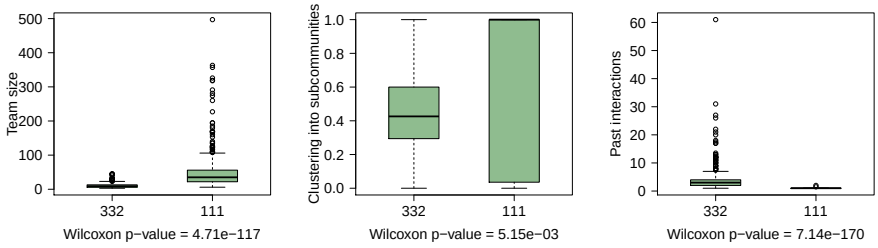
comparison test returns a significance value for each pair of team structures. The significance value indicates if one team structure outranks the other.

Finally, we attempt to examine further characteristics of some team structures associated with the highest and lowest performances. Specifically, we examine three aspects of the projects and their relation with the team structures: a) the size of the projects teams, b) the clustering of the projects networks into cliques (using the transitivity metric explained in Table 2), and *c)* the past interactions among the developers by computing the median weights of the edges between the developers (i.e., the nodes). The higher the weight of an edge between two developers, the more interactions the two developers had in the past.

Results. The rankings across the different metrics show that some team structures, such as 3-3-2 and 3-3-1, are associated to higher performance of the pull-based development model, regardless of the investigated metric. Other team structures characterized by both lower density and reciprocity (e.g., 3-1-1, 2-1-1, and 1-1-1) appear towards the bottom of the ranking. Fig. 8 shows the ranking of the team structures based on the 4 performance metrics. The significance tests confirm that some team structures (i.e., 3-3-X) are significantly superior to other team structures, under all performance metrics. Table 6 lists the team structures that are superior in terms of all performance metrics (column 1), in comparison to the team structures shown in column 2.

We observe that the most frequent team structures do not appear high in the rankings obtained based on all the performance metrics. The most frequent team structures (i.e., 3-1-2 and 3-1-1), that characterize loosely connected teams with very high centralization, rank respectively in the middle and towards the bottom of the 4 rankings shown in Fig. 8. Our ranking results also show that the best team structures according to the 4 rankings (i.e., 3-3-2 and 3-3-1) characterize projects where developers form well-connected teams, and where there are core contributors who take charge. Yet, this group of team structures that ranks high in all 4 rankings accounts for only 24.3% of our pool of projects. Based on the information shown in Table 5, the group 3-3-X is associated to the projects that are smaller in terms of the number of developers compared to others, but comparable in terms of the number of lines of code. We conclude that the development team is able to maintain a superior performance in processing the pull requests with a reduced set of developers. As the number of contributors grows such is the case for the team structure 3-1-1, it takes longer to respond, process, and close the pull requests.

We find that the highest (e.g., 3-3-2) and the lowest (e.g., 1-1-1) performing team structures are statistically different in terms of the three investigated aspects. With regards to the size of the development team, the projects that follow the structure 1-1-1 tend to be larger in size, compared to the projects belonging to the structure 3-3-2. As shown in Fig.7-a, the two distributions of the projects are statistically different based on the results of the Wilcoxon-Mann-Whitney test (Mann and Whitney 1947) (p=value < 0.05). Specifically, the median number of developers in the projects associated to the structure 3-3-2 is 9, versus a median of 35 developers in the projects following the structure 1-1-1. Therefore, despite controlling for the effect of size when defining the team structures, some team structures (e.g., 3-3-2) only exist



(a) Team size

(b) Clustering into sub-communities

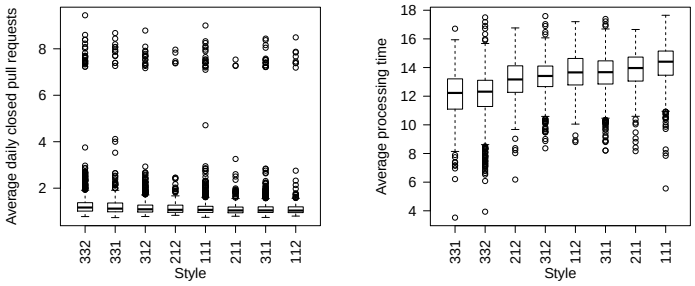
(c) Past interactions

Fig. 7: Significant differences between the high performing team structure 3-1-2 and the low performing team 1-1-1 in terms of 3 aspects

in the smaller projects; possibly because it is not possible to sustain a central core and dense interactions with the increasing number of developers. Second, we measure the clustering into cliques using the transitivity metric (Wasserman and Faust 1994), a measure that varies from 0 when there is no clustering, to 1 for maximal clustering, which happens when the network consists of disjoint cliques. The projects in the first distribution (3-3-2) show moderate clustering (median = 0.42), indicating the existence of some cliques within the network of developers. The second distribution of projects, on the other hand, returns a transitivity median equal to 1, showing that most projects in this distribution exhibit a strong clustering into disjoint cliques (as shown in Fig.7-b). The stronger clustering in projects associated to the structure 1-1-1 could be possibly attributed to different reasons. One possible reason is the modularity of the code base of the projects as they grow, leading to cliques of developers focusing on specific modules of the projects. A second possible explanation is the affinity of developers to work with specific people. Finally, we investigate the difference between the two distributions in terms of the past interactions among the developers. For each project, we compute the median weight of the pull-based network edges. A weight of an edge between 2 developers that is equal to 3 indicates that the two developers interacted 3 times in the past in the review process of the pull requests. We find that the projects associated to the higher performing structure 3-3-2 show higher numbers of past interactions (median = 3), compared to a median of 1 in the second distribution (1-1-1). Therefore, the low performing team structures are more likely to experience the existence of cliques, coupled with the drive-by contributors. This finding agrees with the previous work by Joblin et al. (2017), which also finds evidence regarding the co-existence of cliques and the drive-by contributors. The developers' networks built by Joblin et al. (2017) are based on mailing lists and version control systems.

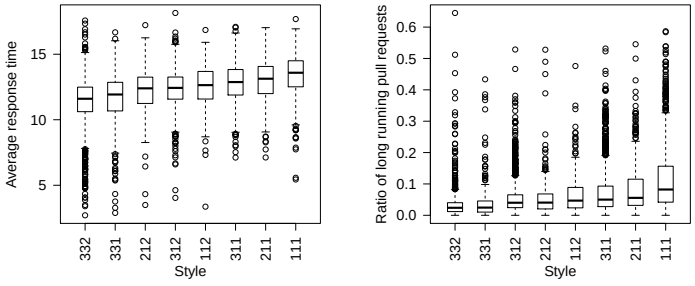
Therefore, we conclude that:

– One of the highest performing team structures (e.g., 3-3-2) only exist in small to medium projects, despite controlling for the effect of size. It is easier for smaller teams to maintain a strong core and dense interactions, and thus achieve better processing of the pull requests.



(a) Average number of pull requests closed daily (higher is better)

(b) Average processing time (lower is better)



(c) Average response time (lower is better)

(d) Ratio of long running pull requests (lower is better)

Fig. 8: Boxplots showing the ranking of the team structures from best to worst. The team structures are encoded as **O**utdegree-**D**ensity-**R**eciprocity

- The formation into disjoint cliques is more present in projects associated with lower performance in processing the pull requests. Projects that maintain a moderate formation of cliques achieve better processing of the pull requests.
- The processing of the pull requests could be faster due to the past interactions between the developers (i.e., building of trust) in the structure 3-3-2. However, in team structures such as 1-1-1, integrators receive most contributions from drive-by and possibly unknown contributors (i.e., the median of past interactions is equal to 1) .

> *The projects, characterized with a well-connected, centralized team around core contributors, are associated to higher response, processing and closing of the pull requests.*

RQ4. Does changing the team structure over time have an impact on the performance of the pull-based development model?

Motivation. The team structure of a team might evolve over time. We are interested in studying if the evolution of the team structures has an impact on the performance

Table 6: Summary of the results of the pairwise comparisons. **Group 1** is the set of team structures associated to significantly higher performance compared to the team structures in **Group 2**, in terms of all the performance metrics

Group 1 (higher performing structures)	Group 2 (lower performing structures)
3-3-2	3-1-2, 2-1-2, 1-1-2, 3-1-1, 2-1-1, 1-1-1
3-3-1	3-1-2, 2-1-2, 1-1-2, 3-1-1, 2-1-1, 1-1-1
3-1-2	3-1-1, 2-1-1, 1-1-1
2-1-2	3-1-1, 2-1-1, 1-1-1
1-1-2	1-1-1
3-1-1	1-1-1
2-1-1	1-1-1

Table 7: The number of GitHub projects with an improvement, deterioration, insignificant change, or no change in terms of the team structure

Evolution type	6 months	1 year	1.5 years	2 years
Improvement	389 (4.95%)	452 (5.76%)	530 (6.75%)	715 (9.11%)
Deterioration	106 (1.35%)	131 (1.67%)	176 (2.24%)	240 (3.06%)
Insignificant change	212 (2.70%)	359 (4.57%)	472 (6.01%)	380 (4.84%)
No change	7,143 (90.99%)	6,908 (88.00%)	6,672 (84.99%)	6,515 (82.99%)

of the pull-based development model. We want to examine if an improvement in the team structures is linked to an improvement in the performance of processing the pull requests.

Approach. First, we collect the GitHub snapshots at different points in time as explained in Section 4.1. Second, for each project, we build a pull-based network based on each temporal snapshot; thus capturing the evolution of the pull-based networks over time. Third, for each project network at time t, we assign the associated team structure based on the network metrics. Lastly, we compute the performance metrics (listed in Section 3.3) of each project at the different points in times to assess the improvement (or decline) of a project in processing pull requests.

To address this question, we use different time intervals (0.5, 1, 1.5, and 2 years) to study the impact of the team structures evolution on the performance of the pull-based development model. Therefore, given an interval $[t_i, t_{i+\Delta}]$, we first decide whether there was an improvement, deterioration, or no change in the evolution of the team structure. Second, we examine whether the performance of the pull-based development model has improved, deteriorated or has not changed between t_i and $t_{i+\Delta}$.

Evolution of the performance metrics:

For each performance metric, we use the values at times t_i and $t_{i+\Delta}$ to measure the change.

1. **Improvement**: If the value at time $t_{i+\Delta}$ is X% better than the value at time t_i, we consider that an improvement has occurred in terms of the given metric. We conjecture that projects of different sizes are able to achieve different efficiency and productivity levels. Therefore, we define X based on the median improvement of similarly-sized projects.

Table 8: Results of Fisher's test and Odds ratio. **P1**: Ratio of long running pull requests, **P2**: Average number of pull requests closed daily, **P3**: Average response time, **P4**: Average processing time

Interval	Impact of improved team structure on the performance			
	P1	**P2**	**P3**	**P4**
	OR (p-value)	OR (p-value)	OR (p-value)	OR (p-value)
6 months	7.22 (1.37e-12)	Inf (2.91e-04)	7.04 (2.75e-11)	5.98 (7.01e-09)
1 year	5.03 (6.36.e-05)	3.75 (1.12.e-04)	3.13 (9.18.e-06)	6.45 (7.12e-08)
1.5 years	6.17 (5.56.e-10)	2.12 (4.58e-08)	2.86 (3.45e-06)	5.12 (8.08e-10)
2 years	4.86 (7.08e-11)	2.85 (3.54e-09)	2.12 (1.12e-08)	5.69 (8.12e-08)

Interval	Impact of deteriorated team structure on the performance			
	P1	**P2**	**P3**	**P4**
	OR (p-value)	OR (p-value)	OR (p-value)	OR (p-value)
6 months	3.61 (2.11e-05)	1.76 (n.s)	3.69 (1.85e-06)	2.48 (1.89e-04)
1 year	1.22 (n.s)	2.15 (3.12.e-04)	1.55 (1.36.e-03)	1.78 (2.15e-03)
1.5 years	1.64 (3.67.e-03)	1.13 (n.s)	1.63 (9.12.e-04)	2.07 (5.64e-06)
2 years	1.78 (2.16e-03)	1.25 (n.s)	1.31 (n.s)	1.55 (3.69e-03)

2. **Deterioration**: If the value at time $t_{i+\Delta}$ is X% worse than the value at time t_i, we consider that a deterioration has occurred in terms of the given metric. X is set based on the median decline of projects of similar sizes.
3. **Constant**: Otherwise, we consider that no change has happened in terms of the performance metric.

Evolution of the team structure:

1. **Improvement** or **deterioration**: In RQ3, we find that some team structures are associated to significantly higher performance than others at a given time t (see Table 6). If the team structure at time $t_{i+\Delta}$ belongs to group 1 in Table 6 (e.g., 3-3-2) and the team structure at time t_i belongs to the list of structures with significantly lower performance in Group 2 (e.g., 1-1-1), we consider the change as an **improvement**. If, on the other hand, the team structure changes the other way (e.g., from 1-1-1 to 3-3-2), we consider the change as a **deterioration**.
2. **Insignificant change**: If the structure changes but the associated performances are not significantly different based on the findings of RQ3 (e.g., 3-3-2 and 3-3-1), we cannot decide whether there is an improvement or deterioration, and therefore, we do not include such instances in this experiment.
3. **Constant**: If the team structure has remained the same (3-3-1 at both t_i and $t_{i+\Delta}$), we consider that there is no change in the team structure.

Next, we examine the impact of a change in the team structure on the performance of the pull-based development model. Accordingly, we define the following null hypotheses:

"H_0^4a: There is no difference in the probability of projects to witness a performance improvement between projects with an improvement or no change in the team structure".

"H_0^4b: There is no difference in the probability of projects to witness a performance deterioration between projects with a deterioration or no change in the team structure".

Table 9: Counts of projects used to compute the Odds Ratio ($OR = \frac{Count_1 * Count_4}{Count_2 * Count_3}$)

		Team structure	
		Improved	Unchanged
Metric P	Improved	$Count_1$	$Count_2$
	Unchanged	$Count_3$	$Count_4$

To test $H_0^4 a$ and $H_0^4 b$, we apply the Fisher's exact test (Sheskin 2007). We reject the null hypothesis if there is statistical significance (i.e., p-value < 0.05). We further compute the Odds Ratio (OR) (Sheskin 2007) to determine if a change in the performance of the pull-based development model has a higher or lower likelihood to occur for projects whose team structure has evolved. To compute the OR within a duration Δt with respect to a performance metric P, we use the counts shown in Table 9. For instance, $count_1$ is the number of projects that witnessed both an improvement in the performance metric P, and an improvement in terms of the team structure within the duration Δt.

Results.

The majority of the projects (82.99% after 2 years) do not witness a change in terms of the team structure. We show in Table 7 the number of projects that witness an improvement, deterioration, insignificant change, or no change in terms of the team structure. Only 9.11% and 3.06% witness a positive and negative change, respectively, after 2 years. 4.84% of the project experience a non-significant change in terms of their team structures. The low percentage of projects that experience a change towards a significantly better team structure might be an indication that the change is not necessarily a result of the natural evolution of projects. Nevertheless, it is not possible to conclude from the data at hand whether the change is conscious or coincidental.

The improvement of the team structure shows strong association with the improvement of the performance of managing the pull requests. The better team structures characterize well-connected teams that are centralized around core contributors. We find that it is highly likely for a development team whose pull-based network shows more desirable properties to improve its efficiency and productivity in managing the pull requests. Our findings (shown in Table 8) are consistent across the different time intervals and across the four performance metrics. For instance, the average processing time of a project is ≈ 6 times likely to decrease after 6 months if the team structure is significantly improved. Therefore, we reject the null hypothesis $H_0^4 a$, and conclude that there is a strong association between the team structures and the performance of the pull-based development model.

A worsening of the team structure is associated in many cases to a deterioration in the performance of managing the pull requests over time. In cases where the team structure of a project deteriorates (i.e., displays higher sparsity and lower centralization around core contributors), we observe the likelihood of a decline in most performance metrics. In some cases (shown in Table 8), the difference is not significant (i.e., p-value > 0.05 and OR < 1.5), indicating no strong association

between the deterioration in the team structure and the performance (especially for the average number of pull requests closed daily). The overall results lead us to reject the null hypothesis $H_0^4 b$, and conclude that it is likely for a team to witness a decline in the efficiency and productivity of managing the pull requests when the density and centralization of the pull-based network decreases. We further inspect the impact of an increasing size of the development team on the performance metrics. We find a stronger association between the increasing number of developers and the increasing processing time of the pull requests, compared to the impact of a deteriorating team structure on slowing down the processing time of pull requests. For instance, pull requests are processed 8.96 times slower (p-value = 2.45e-18) with an increasing number of developers, over a period of 2 years.

We find that the move of a development team towards a structure that is more centralized (i.e., the presence of a set of core contributors), more dense (i.e., each developer interacts with a larger set of developers), and more reciprocal (i.e., more developers are involved as both integrators and contributors), is associated with a significant improvement in the speed of processing the pull requests.

6 Discussion

Summary of Contributions

To gain a better understanding on the network metrics that can possibly impact the processing of the pull requests, we study the relationship between a set of network metrics extracted from the pull-based networks (e.g., reciprocity) and a set of performance metrics (e.g., response time). The results of the linear regression reveal that the network metrics can partially explain the performance metrics (R^2 <= 0.32). Specifically, maintaining a strong core of contributors and denser interactions among the developers is associated with faster response and processing of the pull requests. The remaining variability observed in the models is likely due to other factors, such as the churn of the code change associated with the pull requests, the expertise of the reviewers, and the turnover of the developers. Although the models built in our study do not incorporate non-network based metrics, the network metrics, such as centralization, implicitly cover some of the unaccounted for aspects. For instance, if a core member exits a project (as a result of turnover), a drop in the centralization of the project network would likely be observed. Another example is that developer's expertise is possibly reflected through their number of contributions (i.e., centralization), and their ability to both produce and review pull requests (i.e., reciprocity). However, more comprehensive models (i.e., that include both network and non-network measures) could be essential to fully explain the performance of the pull-based model.

Using the significant network metrics, we systematically define and identify a set of team structures, and examine their popularity in our subject projects. We observe that the most commonly followed group of structures (Group 2 in RQ2) is characterized by

sparse collaboration links among the developers (i.e., low density), and strict decision making process (i.e., low reciprocity). Besides, it appears that smaller and larger projects form into different team structures. Therefore, we believe that certain team behaviors are harder to sustain in larger projects (e.g., high centrality coupled with dense collaborations - Group 3 in RQ2).

A comparison of the different team structures reveals that the projects that maintain a set of core contributors that collaborate densely achieve better performance in processing the pull requests. We take a closer look into the better performing teams structures. We observe that developers within such projects have a higher median of past interactions, as measured by the weights of edges in the pull-based networks. This finding confirms the findings of prior work related to the pull request evaluation (Tsay et al. 2014a; Gousios et al. 2014). Indeed, Tsay et al. (2014a) report that prior contributions are used as a signal of the trustworthiness of a contributor; while Gousios et al. (2014) conclude that the time to merge a pull request is influenced by a contributor's previous track record.

Socially-enabled governance in open source projects: what's different?

Many of our findings confirm the results of previous studies that look into the structural properties of open source projects, based on different artifacts (e.g., bug tracking systems or mailing lists). However, the structural properties of teams as they collaborate using pull requests have not been studied in the prior work. The pull request development model is unique in its nature, because it happens in the context of the social coding model. Specifically, thanks to the transparency available in platforms such as GitHub, it is easy for the developers to collect social signals about each others, such as, the size of their followers, their number of stars, and their activity overview (e.g., 54% of the developer's activity is code review and 10% is issue reports). This results in a unique environment of collaboration, where both technical and social factors come in play during the process of code (and developer) evaluation (Tsay et al. 2014a). Therefore, it is important to re-examine the commonly known principles of open source governance, in a socially-enabled collaboration environment. Our findings suggest similarities with traditional collaboration venues, such as bug tracking systems. Moreover, it reveals interesting insights, such as, how much can structural properties explain the performance of the pull-based model, the distribution of the identified team structures and the characteristics of their associated projects, the trends of evolution of the structures over time, and the possible warning signs that a project is evolving towards a worsening structure.

Do projects evolve naturally over time to form better team structures?

Another question that arises from our study is whether the shift of projects towards more desired team structures a result of natural evolution, or a conscious choice made by the maintainers of projects. In our study, we find that:

a) only a low percentage of the projects witnesses a change towards team structures with the desired properties (e.g., stronger centralization) over time,

b) the most efficient and productive team structures (e.g., 3-3-2) are associated only to the smaller sized projects, where the size is measured using of the number of the developers.

The two aforementioned findings lead us to believe that projects do not evolve naturally into team structures that achieve better performance of processing the pull requests. As such, we recommend that the maintainers of the open source projects consciously monitor the evolution of their team structures in the pull-based development model.

How can maintainers monitor and improve the structure of their teams?

The pull-based networks can potentially help identify the overwhelmed developers (i.e., the developers with high centralization scores), or the developers with similar knowledge (to select the developers for sharing or shifting the development tasks). The maintainers of the projects could possibly prevent the evolution into team structure associated with low performance. One of the warnings signs that a team structure is evolving negatively is the formation of disjoint cliques as discussed in RQ3. Another warning sign is the increase in the number of the 'drive-by' contributors, coupled with a decrease in the presence of the core and trusted contributors. It would be possible in practice to prevent the propagation of this phenomena by:

- creating mentoring or explicit links between new contributors and established developers within the projects, to built trust and favor retention. The creation of links involves assigning specific developers to review the pull requests submitted by new and unknown contributors. The assigned developers would ideally provide assistance to the new contributors, and build their familiarity with the norms and conventions of the projects.
- maintaining a central core within the team despite growing in size. To avoid the extensive formation of cliques, it is important for a set of developers to maintain a general awareness of the pull requests submitted in a project. Such developers would periodically get involved in the review process of pull requests associated to different modules in the project.

7 Threats to Validity

This section discusses the threats to validity of our study.

Threats to conclusion validity concern the relation between the treatment and the outcome. The performance in managing pull requests is impacted by many factors, other than the team structure, such as the turnover of the developers, and the amount and complexity of pull requests received by the project. Therefore, the performance of the pull-based model cannot be fully explained by one of the factors only. However, as discussed in Section 6, we are still able to obtain interesting insights from the relationship between the team structures and the performance of the pull-based development model. Our future work will consider additional factors, such as the amount and quality of pull requests received by the project, to properly model the performance of the pull-based development model.

Threats to internal validity concern our selection of subject projects and analysis methods. The completeness and popularity of the team structures identified depends on the selected pool of projects. Since we study the pull-based networks, we select the projects with the highest number of recorded pull requests (over 100 pull requests). To

ensure the stability of our results, we vary the threshold of pull requests used to select the projects, and find that the most frequent team structures are similar with different numbers of selected projects. As shown in Table 1, the majority of our subject projects (69.8%, i.e., 5,483 out of 7,850) have 20 or less developers. This could indicate that our dataset could be biased towards the smaller projects. However, as reported by Gousios et al. (2014), around 70% of the projects hosted on GitHub are single-developer projects. As such, the resulting set of projects only mirrors the widespread of the smaller projects hosted on GitHub, in terms of the number of developers. Besides, our study design attempts to control for size using both the CUG test for normalization, and by including control metrics (i.e., the number of developers and the number of commits) in the models built. In our study, we consider the evolution of the team structures using temporal snapshots of the projects separated by at least 6 months. We do not study duration less than 6 months because we observe that 5% or less of the projects experience a change in the team structure in shorter durations.

Threats to external validity concern the possibility to generalize our results. GitHub is the number one social coding platform with millions of active repositories. We study the 7,850 most active projects, and therefore our findings are based on a very active pool of projects. Besides, our findings can be generalized to other open source projects using different social coding platforms, such as BitBucket, because the pull-based development model works the same across platforms.

8 Conclusion

In open source distributed projects, the collaboration and self-organization of developers is an important aspect to the success of projects. With the social coding platforms, distributed teams are able to collaborate using friendly features, such as pull requests. Through a single place (i.e., the pull request), code can be proposed, reviewed, and safely merged to the main stream of a project. However, the pull-based development model exhibits to challenges, such as the ability of the team to process all incoming pull requests on a timely manner. The faster pull requests are merged into a project, the more issues are fixed or the more new features are introduced to the project.

In this paper, we build pull-based networks for the most popular projects on GitHub and propose a way to identify the existing team structures based on a set of influential network metrics. We find that over a third of the projects are characterized by loosely connected teams, with single-role developers. Then, we attempt to rank the different team structures and find that the most desirable structure characterizes projects where developers are well connected (i.e., developers collaborate with many (if not most) of their teammates), are centralized around key contributors, and possibly take roles as both integrators and contributors. Finally, we study the evolution of the team structures along with the performance of the projects in processing the pull requests. Our findings reveal a strong association between the improvement in the team structure and the increase in the performance of the pull-based development model. In the future, we plan to study the team structures in more social coding platforms, and include additional factors that could impact the performance of the pull-based development model.

References

Anderson BS, Butts C, Carley K (1999) The interaction of size and density with graph-level indices. Social Networks 21(3):239–267

Barr ET, Bird C, PC, Hindle A, German DM, Devanbu P (2012) Cohesive and isolated development with branches. In: Fundamental Approaches to Software Engineering, Springer, pp 316–331

Bersani FS, Lindqvist D, Mellon SH, Epel ES, Yehuda R, Flory J, Henn-Hasse C, Bierer LM, Makotkine I, Abu-Amara D, Coy M, Reus VI, Lin J, Blackburn EH, Marmar C, Wolkowitz OM (2016) Association of dimensional psychological health measures with telomere length in male war veterans. Journal of Affective Disorders 190:537 – 542

Bettenburg N, Hassan AE (2010) Studying the impact of social structures on software quality. In: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, pp 124–133

Bird C, Gourley A, Devanbu P, Swaminathan A, Hsu G (2007) Open borders? immigration in open source projects. In: Proceedings of the Fourth International Workshop on Mining Software Repositories, IEEE Computer Society, MSR '07, pp 6–

Bird C, Pattison D, D'Souza R, Filkov V, Devanbu P (2008) Latent social structure in open source projects. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, ACM, pp 24–35

Butts CT, et al. (2008) Social network analysis with sna. Journal of Statistical Software 24(6):1–51

Capra E, Francalanci C, Merlo F (2008) An empirical study on the relationship between software design quality, development effort and governance in open source projects. IEEE Transactions on Software Engineering 34(6):765–782

Choi H, Varian H (2012) Predicting the present with google trends. Economic Record 88:2–9

Crowston K, Howison J (2005) The social structure of free and open source software development. First Monday 10(2)

Crowston K, Howison J (2006) Hierarchy and centralization in free and open source software team communications. Knowledge, Technology & Policy 18(4):65–85

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, ACM, New York, NY, USA, CSCW '12, pp 1277–1286

Dinh-Trong TT, Bieman JM (2005) The freebsd project: A replication case study of open source development. IEEE Transactions on Software Engineering 31(6):481–494

Ducheneaut N (2005) Socialization in an open source software community: A socio-technical analysis. Computer Supported Cooperative Work (CSCW) 14(4):323–368

Ehrlich K, Cataldo M (2012) All-for-one and one-for-all?: A multi-level analysis of communication patterns and individual performance in geographically distributed software development. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, ACM, New York, NY, USA, CSCW '12, pp 945–954

Freeman LC (1977) A set of measures of centrality based on betweenness. Sociometry pp 35–41

Freeman LC (1978) Centrality in social networks conceptual clarification. Social networks 1(3):215–239

Gacek C, Arief B (2004) The many meanings of open source. IEEE software 21(1):34–40

Garlaschelli D, Loffredo MI (2004) Patterns of link reciprocity in directed networks. Physical review letters 93(26):268,701

Gharehyazie M, Posnett D, Vasilescu B, Filkov V (2015) Developer initiation and social interactions in oss: A case study of the apache software foundation. Empirical Software Engineering 20(5):1318–1353

Gousios G (2013) The ghtorent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, pp 233–236

Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 345–355

Gousios G, Zaidman A, Storey MA, van Deursen A (2015) Work practices and challenges in pull-based development: The integrator's perspective. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 358–368

Gousios G, Storey MA, Bacchelli A (2016) Work practices and challenges in pull-based development: The contributor's perspective. In: Proceedings of the 38th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '16, pp 285–296

Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2008) statnet: Software tools for the representation, visualization, analysis and simulation of network data. Journal of statistical software 24(1):1548

Howison J, Inoue K, Crowston K (2006) Social dynamics of free and open source team communications. In: IFIP International Conference on Open Source Systems, Springer, pp 319–330

Jiang Y, Adams B, German DM (2013) Will my patch make it? and how fast?: Case study on the linux kernel. In: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, Piscataway, NJ, USA, MSR '13, pp 101–110

Joblin M, Apel S, Hunsen C, Mauerer W (2017) Classifying developers into core and peripheral: An empirical study on count and network metrics. In: Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on, IEEE, pp 164–174

Krackhardt D (1994) Graph theoretical dimensions of informal organizations. Computational organization theory 89(112):123–140

von Krogh G, Spaeth S, Lakhani KR (2003) Community, joining, and specialization in open source software innovation: a case study. Research Policy 32(7):1217 – 1241, open Source Software Development

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. Journal of the American statistical Association 47(260):583–621

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp 50–60

Marlow J, Dabbish L, Herbsleb J (2013) Impression formation in online peer production: Activity traces and personal profiles in github. In: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, ACM, New York, NY, USA, CSCW '13, pp 117–128

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

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '08, pp 181–190

Nagappan N, Ball T (2007) Using software dependencies and churn metrics to predict field failures: An empirical case study. In: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, Washington, DC, USA, ESEM '07, pp 364–373

O'Mahony S, Ferraro F (2007) The emergence of governance in an open source community. Academy of Management Journal 50(5):1079–1106

de Reus MA, van den Heuvel MP (2013) The parcellation-based connectome: Limitations and extensions. NeuroImage 80:397 – 404, mapping the Connectome

Rick (2013) View long-running pull requests

Rigby PC, Storey MA (2011) Understanding broadcast based peer review on open source software projects. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 541–550

Rigby PC, Barr ET, Bird C, Devanbu P, German DM (2013) What effect does distributed version control have on oss project organization? In: Release Engineering (RELENG), 2013 1st International Workshop on, IEEE, pp 29–32

Robertsa J, Hann IH, Slaughter S (2006) Communication networks in an open source software project. In: IFIP International Conference on Open Source Systems, Springer, pp 297–306

Schall D (2014) Who to follow recommendation in large-scale online development communities. Information and Software Technology 56(12):1543 – 1555, special issue: Human Factors in Software Development

Sheskin DJ (2007) Handbook of Parametric and Nonparametric Statistical Procedures, 4th edn. Chapman & Hall/CRC

Siegel S (1956) Nonparametric statistics for the behavioral sciences. McGraw-hill

Steel RGD, Torrie JH (1960) Principles and procedures of statistics: with special reference to the biological sciences. McGraw-Hill

Tsay J, Dabbish L, Herbsleb J (2014a) Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 356–366

Tsay J, Dabbish L, Herbsleb J (2014b) Let's talk about it: Evaluating contributions through discussion in github. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2014, pp 144–154

Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp 805–816

Wasserman S, Faust K (1994) Social network analysis: Methods and applications, vol 8. Cambridge university press

Wolf T, Schroter A, Damian D, Nguyen T (2009) Predicting build failures using social network analysis on developer communication. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 1–11

Yu Y, Wang H, Yin G, Ling CX (2014a) Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration. In: 21st Asia-Pacific Software Engineering Conference, vol 1, pp 335–342

Yu Y, Yin G, Wang H, Wang T (2014b) Exploring the patterns of social behavior in github. In: Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies, ACM, New York, NY, USA, CrowdSoft 2014, pp 31–36

Yu Y, Wang H, Filkov V, Devanbu P, Vasilescu B (2015) Wait for it: Determinants of pull request evaluation latency on github. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp 367–371

Zanetti MS, Scholtes I, Tessone CJ, Schweitzer F (2013) Categorizing bugs with social networks: A case study on four open source software communities. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 1032–1041

Zar JH (2005) Spearman Rank Correlation, John Wiley & Sons, Ltd

Mariam El Mezouar is a PhD candidate in the School of Computing at Queen's University (Canada). She received her B.S degree in Computer Science and her M.S degree in Software Engineering from Al Akhawayn University (Morocco) in 2011 and 2013 , respectively. Her research interests include software engineering, mining software repositories and defect prediction.

Feng Zhang is currently a Ph.D. candidate in School of Computing from Queen's University in Canada. He received both his B.S. degree and his M.S. degree from Nanjing University of Science and Technology (China) in 2004 and 2006, respectively. His research interests include empirical software engineering, software re-engineering, mining software repositories, source code analysis, and defect prediction. More about Feng and his work is available online at http://feng-zhang.com

Ying Zou is a Canada Research Chair in Software Evolution. She is an associate professor in the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture.