Hierarchical and Hybrid Organizational Structures in Open-Source Software Projects: A Longitudinal Study

MITCHELL JOBLIN, Saarland University & Siemens, Germany
BARBARA ECKL-GANSER, University of Passau, Germany
THOMAS BOCK, Saarland University, Saarland Informatics Campus, Germany
ANGELIKA SCHMID, IBM, Germany
JANET SIEGMUND, Chemnitz University of Technology, Germany
SVEN APEL, Saarland University, Saarland Informatics Campus, Germany

Despite the absence of a formal process and a central command-and-control structure, developer organization in open-source software (OSS) projects is far from being a purely random process. Prior work indicates that, over time, highly successful OSS projects develop a *hybrid* organizational structure that comprises a *hierarchical* part and a *non-hierarchical* part. This suggests that hierarchical organization is not necessarily a global organizing principle and that a fundamentally different principle is at play below the lowest positions in the hierarchy. Given the vast proportion of developers are in the non-hierarchical part, we seek to understand the interplay between these two fundamentally differently organized groups, how this *hybrid structure* evolves, and the trajectory individual developers take through these structures over the course of their participation. We conducted a longitudinal study of the full histories of 20 popular OSS projects, modeling their organizational structures as networks of developers connected by communication ties and characterizing developers' positions in terms of hierarchical (sub)structures in these networks. We observed a number of notable trends and patterns in the subject projects: (1) hierarchy is a pervasive structural feature of developer networks of OSS projects; (2) OSS projects tend to form hybrid organizational structures, consisting of a hierarchical and a non-hierarchical part; and (3) the positional trajectory of a developer starts loosely connected in the non-hierarchical part and then tightly integrate into the hierarchical part, which is associated with acquisition of experience (tenure), in addition to coordination and coding activities. Our study (a) provides a methodological basis for further investigations of hierarchy formation, (b) suggests a number of hypotheses on prevalent organizational patterns and trends in OSS projects to be addressed in further work, and (c) may ultimately guide the governance of organizational structures.

CCS Concepts: • **Software and its engineering → Software creation and management**; **Collaboration in software development**; **Open source model**.

Additional Key Words and Phrases: open-source software projects, developer networks, organizational structure, hierarchy

ACM Reference Format:
Mitchell Joblin, Barbara Eckl-Ganser, Thomas Bock, Angelika Schmid, Janet Siegmund, and Sven Apel. 2022. Hierarchical and Hybrid Organizational Structures in Open-Source Software Projects: A Longitudinal Study. *ACM Trans. Softw. Eng. Methodol.* , (October 2022), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Mitchell Joblin, Saarland University & Siemens, Germany; Barbara Eckl-Ganser, University of Passau, Germany; Thomas Bock, Saarland University, Saarland Informatics Campus, Germany; Angelika Schmid, IBM, Germany; Janet Siegmund, Chemnitz University of Technology, Germany; Sven Apel, Saarland University, Saarland Informatics Campus, Germany.

1049-331X/2022/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

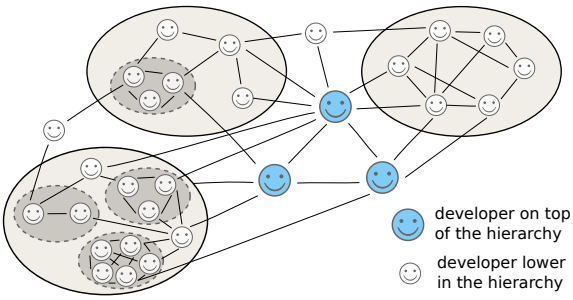

Fig. 1. Even without formal process or mandated developer roles, hierarchical structure emerges in OSS projects, with few developers on top of the hierarchy (blue, big smileys) and many developers lower in or even outside the hierarchy (gray, small smileys) [25].

1 INTRODUCTION

Many software systems are developed either as open-source software (OSS) or rely on OSS libraries, frameworks, etc. [47]. OSS developers actively contribute to an OSS project by means of coding and by issuing and discussing bug reports, feature requests, etc. OSS developers typically organize in a decentralized and self-organized manner [4]. Despite the absence of a formal process and a central command-and-control structure, prior studies have shown that developer organization is far from being a purely random process. Studies of developer organization based on network models have been accumulating growing evidence that multiple organizing principles are simultaneously at play [4, 25–27]. Two important organizing principles are: (1) the probability that a randomly selected developer has k connections to other developers in the network is described by a power law—the *scale-freeness principle*—and (2) developers organize into densely connected groups—the *community principle*. It has been shown that, to simultaneously reconcile these two principles under one roof, the groups must arrange according to a hierarchy (see Fig. 1) [39]. Hierarchical organization induces a dependence between the number of connections to a developer (node degree) and the density of local connections (clustering coefficient), which can be used to test for the presence of hierarchy [39].

Recently, Joblin et al. [26] found indications that, over time, highly successful OSS projects develop a *hybrid* organizational structure that comprises a *hierarchical* part and a *non-hierarchical* part. This observation is consistent with previous findings reported in the literature [11, 33] and is intriguing for two reasons: (1) it implies that hierarchical organization is not necessarily a global organizing principle and that a fundamentally different principle is at play below the lowest positions in the hierarchy; (2) the vast majority of developers often occupy non-hierarchically organized positions in the developer network. In this vein, previous work suggests a connection between role stratification and the emergence of organizational structure [26]. *Role stratification* is the process of the emergence of developer roles arising from differences in the developers' behavior. Role stratification limits coordination overhead and improves information flow, this way, influencing project quality and performance [18, 26, 28, 36, 45, 46]. In particular, Joblin et al. [25] provide evidence that developer roles and hierarchy in developer networks are connected such that *core developers occupy upper positions* in the hierarchy and *peripheral developers occupy lower positions* [25], where *core developers* play an essential role in developing the system architecture, programming, and forming the general leadership structure, with long-term involvement, and (2) *peripheral* developers help with bug fixes or small enhancements, with irregular or short-term involvement.

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

To address these questions, we conduct a longitudinal study on 20 popular OSS projects of various application domains and sizes with a total of 831 6-months snapshots. We explore their organizational structure as a network of developers who are connected by communication ties [29]. Based on these developer networks, we divide the set of developers into a hierarchical and non-hierarchical part and explore its evolutionary trends (RQ_1), we track the neighborhood of individual developers to understand their placement and progression in the hierarchy (RQ_2, RQ_3), and we characterize the roles of developers in the hierarchy with information on tenure and activity (RQ_4).

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

In summary, we make the following contributions:

- insights regarding the presence and evolution of hierarchical and hybrid organizational structures in OSS projects and a method to identify these structures;
- findings regarding the nature of the relationship between members of the hierarchical and non-hierarchical parts;
- a comparison of the developers in the hierarchical and non-hierarchical parts regarding tenure and activity level;
- a discussion of practical implications and hypotheses that shall guide further research;
- full access to our analysis scripts and results on a supplementary website: https://hierarchypaper.bitbucket.io/public/ and https://zenodo.org/record/7199267.

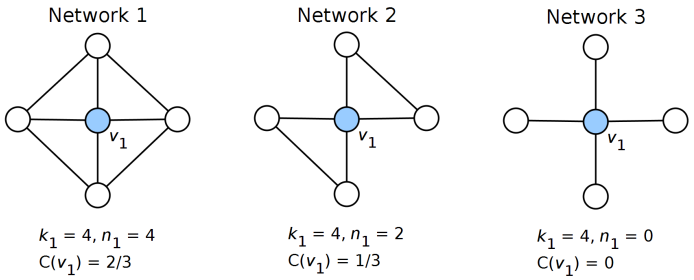

Fig. 2. Three networks with the corresponding clustering coefficient $c_i = 2 \cdot n_i/(k_i \cdot (k_i - 1))$ for node v_1 in descending order according to clustering coefficient (left to right).

2 BACKGROUND & RELATED WORK

In this section, we provide general background information and related work on network hierarchy and other network concepts. Thereafter, we discuss related work on developer networks specifically.

2.1 Network Analysis

A network is an object consisting of a set of nodes and a set of binary relations or links between the nodes. The most common type of network is the homogeneous, undirected, and unweighted network which consists of only one type of node, the link implies a symmetric relationship, and each link is defined to have equivalent magnitude. We use this network type in our study. Given a network, there exists a suite of analysis metrics for studying its properties. Two fundamental node-level metrics are the degree and clustering coefficient. A node's *degree* is defined to be the number of links it appears in. For example, in Fig. 2, the node labeled v_1 has a degree of four. While the node degree captures information between a node and its immediate neighborhood, the node *clustering coefficient* captures information about how members of a node's neighborhood are connected to each other. More specifically, the clustering coefficient is the ratio of existing links to all possible links among a node's direct neighbors. The clustering coefficient c_i is defined as $2 \cdot n_i/(k_i \cdot (k_i - 1))$, with n_i being the number of links between the k_i neighbors of node v_i. A fully connected subgraph has a clustering coefficient of 1. Nodes with only one neighbor have a degree of 1, but no valid clustering coefficient. Examples of various networks and corresponding clustering coefficients are illustrated in Fig. 2.

Depending on the underlying organizational principles that influence the formation of links in a network, the types of structural features the networks posses can differ significantly. For example, if the existence of each link is determined purely by chance (i.e., by flipping an unbiased coin where heads corresponds to a link and tails does not) then an Erdős and Rényi (ER) random network is generated as shown in Fig. 3 (left) [13]. Due to the independent formation of each link, these networks lack higher-order structure (e.g., communities or hierarchy). To achieve a departure from these purely random network structures, an underlying organizing principle must induce a dependence between the links. For example, if groups of nodes exist such that forming links among members of the same group is more likely than forming links with members of different groups, then higher-order structure in the form of communities arises [14]. Some real-world networks are also known to be scale-free, which implies that the degree distribution of nodes follows a power-law distribution [35]. One way this property occurs is through a dependence between degree and the probability of link formation, such that nodes with a higher degree are more likely to gain new links than a node with a lower degree, which is known as preferential attachment [35].

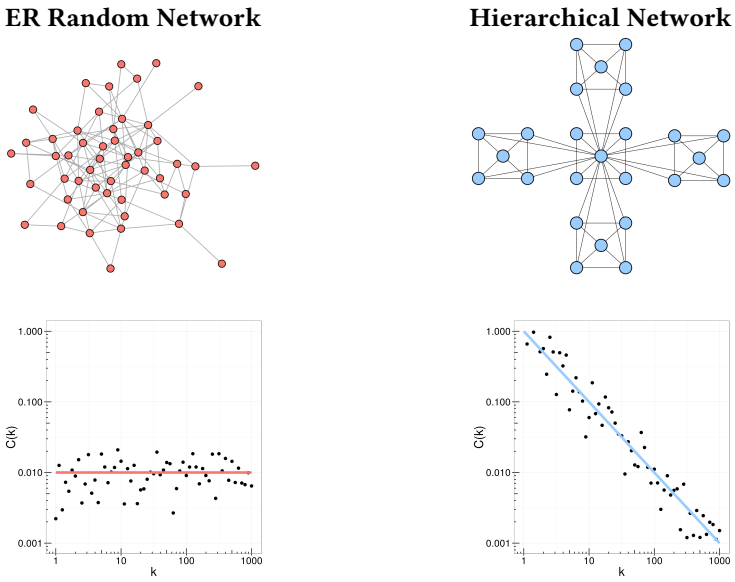

Fig. 3. Random network (left) and hierarchical network (right) and below a scatter plot of node degree k versus clustering coefficient $C(k)$ for each network.

A third example of non-random structure is hierarchy. Hierarchical structure in networks can be achieved by inducing a dependence between the clustering coefficient and the degree [39]. Network hierarchy manifests as local clustering within a global tree-like arrangement of these clusters, which is shown in Fig. 3 (right). Due to the stochastic nature of complex networks, the concept of hierarchy differs slightly from a strict hierarchy which forbids relationship between entities at the same layer. It should be noted that hierarchy is distinct from community and core/periphery structures which do not require a dependence between the clustering coefficient and degree [39].

2.2 Developer Networks

For studying socio-technical aspects of software development, networks are used to represent relationships between developers induced by their development activities. Typically, networks are constructed by considering information from the version control system by extracting relationships from the commit activity [7–9, 25–27, 31, 34], from communication activity through issue trackers or mailing lists [4, 20, 49], or a combination of both [3, 5, 30].

Network representations of software projects have proven to be a powerful abstraction in numerous applications. Wolf et al. [49] showed that developer networks can successfully predict the occurrence of build failures. Multiple studies have demonstrated that developer networks can successfully predict software defects [3, 32] and Nagappan et al. [34] found that organizational metrics are even more predictive of software quality factors than traditional source code metrics. Shin et al. [41] showed that developer activity metrics extracted from networks are predictive of security vulnerabilities. When it comes to developers roles, Joblin et al. [25] found that the structural position of developers within the network is reflective of developer perception. These studies show evidence that developer networks contain rich structure that is related to several highly relevant socio-technical dimensions of a software project. Inspired by these results, our

primary focus is on achieving a deeper understanding what fundamental structures exist, how they evolve with time, and how developers embed within them.

Determining whether developer networks' link formation is driven primarily by a purely random process or by non-random organizational principles has received some attention. Bird et al. [4] identified that developer communication networks contain a latent community structure that is consistent with non-random link formation. Similarly, in developer networks constructed from commit activity, Joblin et al. [27] found that developers form communities that are statistically significant and align with developer's perception of team collaboration. The scale-free property also appears to be pervasive among projects that achieve sustained long-term growth and there is initial evidence of hierarchy existing in OSS projects [26]. Given that hierarchy functions as a unifying principle of two other important structures, communities and scale-freeness, it is imperative to better understand the role hierarchy plays is OSS projects.

3 RESEARCH QUESTIONS

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

RQ_1: Are there patterns in terms of structure and evolution that exist in OSS projects with respect to hierarchy?

Hierarchy and role stratification emerge in times of project growth and increased coordination overhead [21, 45, 50], and organizational structure is closely related to project sustainability and scalability [51]. As a consequence, the fundamental organizing principles at play must evolve throughout the project life cycle. It remains an open question, however, whether hierarchical, non-hierarchical, and hybrid structures are a pervasive phenomenon governing organization in OSS projects and at which points in the project lifecycle these structures exist. Answering RQ_1 sheds light on hierarchy and hybrid structure as a global network property that co-evolves with the different project phases and growth regimes. Since each type of organizational structure exhibits certain strengths and weaknesses, by knowing what organizational structures exist and when, we would be able to asses whether the observed structures make appropriate accommodations to the current project conditions.

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

While the perspective of RQ_1 is rather global, we now peer through the lens of individual developers to understand their personal journey through the organizational structure. Similar to the organizational structure of a project, the roles of developers evolve along with their activity in the project [52]. Understanding the social dynamics in OSS projects requires understanding the relationship between on-boarding of developers and their later roles [6, 54] and the determinants of the chances to reach an influential position [10, 53]. In our study, we look for typical "trajectories" of developers through a project's hierarchy. Clearly, different roles imply different experience levels, consistency of involvement, and level of commitment. Since certain organizational structures are not well suited for high volatility (e.g., hierarchy), the inconsistent involvement exhibited by some roles posses a risk, unless the roles are organized differently. By answering this question, we are able to observe how large-scale and successful OSS projects integrate new developers and how their progression is reflected in their structural position. This information is useful to identify problematic on-boarding, off-boarding, and developmental progression practices occurring in a project and apply corrective measures to ensure a higher probability of success.

RQ$_3$: How are a developer's contacts distributed over the organizational structure of an OSS project?

One way how network hierarchy can benefit project quality and success is through improved communication and information flow. Consequently, it is important to understand who a developer's social contacts and cooperation partners are. Canfora et al. [6] investigated who is responsible for on-boarding new community members in OSS projects. They found that whether a developer's first contact is an isolated or a well-integrated community member has an important influence on information flow and knowledge sharing. Afsar and Badir [1] and Zhou and Mockus [55] provide more evidence for knowledge sharing of colleagues and especially newcomers during their on-boarding phase. Steinmacher et al. [42] analyze the extent to which social contacts into the group of an OSS community's core developers simplify on-boarding into new projects. We expect that developers who communicate early with other developers from the top of the hierarchy will eventually rise in the hierarchy, which we will verify by tracking their positions in the hierarchy over time.

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

Since different positions in the organizational structure enable different functions, in a healthy project, the position a developer has should ideally support the developer's role and should not expose the project to unnecessary risk. In OSS projects, important developers are often responsible for both, coordination and the bulk of the programming work [12, 40, 50]. By analyzing e-mail data and data from issue trackers in combination with commit data, we are able to investigate the relationship of coordination and programming tasks with respect to a developer's position in the hierarchical structure of the project. The entire process usually takes time: Developers become core developers and project leaders only with sufficient experience in the project [2, 54]. In combination, these aspects may explain why tenure has only a weak relation to the number of code contributions: The number of code contributions increases only in the starting phase of a project and caps after around three years [52]. It is beneficial to understand the relationship between function and structural position in large-scale OSS projects, because it helps us to establish practices that are conducive to successful outcomes. For example, if developers with primarily coordination tasks are positioned outside of the hierarchy, that could be an indication that there is a mismatch between the function and structural position. In that case, policies and practices should be revised to induce a healthier project structure.

4 METHOD

We now introduce the details for constructing developer networks and the corresponding statistical methods used to identify the hierarchical and non-hierarchical developer portions.

4.1 Data Extraction & Construction of Developer Networks

To create developer networks from OSS projects, we extract link structure based on e-mail communication on the developer mailing list of the project or based on issue discussions on issue trackers corresponding to the project. We downloaded developer mailing lists from the mailing list archive GMANE[1] and retrieved GitHub issue discussions from GitHub's REST API[2] for issue comments,

[1]http://gmane.org/
[2]https://docs.github.com/en/rest/

review comments, and pull-request comments.[3] For the extraction of background information about developers and their commits in an OSS project, we use CODEFACE[4], which includes, beside mining of software repositories and mailing lists, entity matching between the developers in the version control system and the senders and receivers of e-mails or issue discussions.

The developer networks contain only a single node type, denoting a developer, and a single link type, denoting a communication activity, that is, e-mail communication for mail networks and issue communication for issue networks. We define a link to exist between a pair of developers when both have contributed, at least, one e-mail to a common e-mail thread on the mailing list or made a comment [5] to, at least, one common issue. E-mails relating to a common thread can be trivially grouped on the basis of a unique thread identifier, issue comments are automatically related to an issue when extracting them from GitHub's API. Links are undirected and have no weight attribute. Note that the hierarchy measure that we use is defined on undirected networks [39]. To build and analyze the developer networks, we use the library CORONET[6] [23].

We capture the temporal dimension of the project history by applying a sliding window approach to produce a sequence of networks [26]. This means, the communication activity used to construct a developer network spans a time range of six months and the sliding windows overlap by 50% (i.e., an overlap of three months) to allow for smooth transitions between adjacent windows. This choice balances between fluctuation and noise in short time ranges (prohibiting us from observing any non-spurious patterns, plus randomly losing communication activity that happens across time range borders) and losing detail in long ranges due to aggregation [31].

4.2 Typical Structure and Evolution (RQ_1)

For each network, we visualize, per analyzed time range, the hierarchical position of all developers in one plot, as shown in Fig. 4. The plot shows the values k_i (x axis, logarithmic scale) and c_i (y axis, logarithmic scale) for every developer i for one time range. The right side of the dashed line shows the relationship between c_i and k_i that is typical for a hierarchical network: a linear relationship with negative slope [39]. To the left of the dashed line, the relationship appears fundamentally different in that there is certainly no linear relationship and even appears to be no relationship. The absence of a linear relationship with a negative slope indicates that hierarchical organization is not present and is consistent with the random graph introduced in Fig. 3. Determining the hierarchical and non-hierarchical parts corresponds to identifying a breakpoint between the linearly related segment and remaining unrelated segment (i.e., identifying an optimal position for the dotted line in Fig. 4). Fig. 5 illustrates the decision space for this problem, where each vertical line corresponds to a candidate breakpoint. Our method for automating the identification of this breakpoint relies on a mixed approach of human labeling and combinatorial optimization. The human labeling is used to learn rules for trimming the candidate space, making the combinatorial optimization procedure more efficient. As a secondary use of the human labeled data, we test how well our automated method generalizes to decisions made by human annotators that were not used to find parameters for the method.

[3] As previous research has shown that GitHub issue data could be distorted by bots that automatically comment on issues but are not human beings [16, 48], we used the tool BODEGHA [15] to remove comments created by bots from the issue discussions.

[4] https://github.com/se-sic/codeface/

[5] We also treat the initial comment authored by the issue creator as a comment, as well as pull-request review comments and all their replies.

[6] https://github.com/se-sic/coronet/

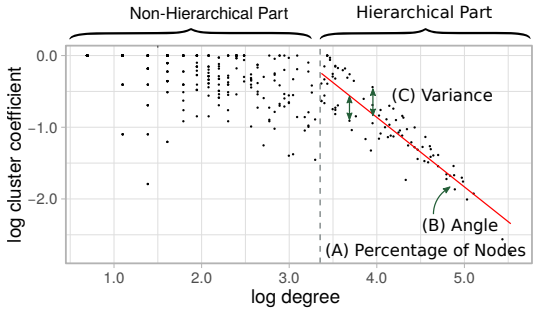

Fig. 4. The plot shows the node degree and clustering coefficient of all developers who were active in this time range. Each dot represents one developer. There is a division into a hierarchical and a non-hierarchical part (dashed line). We mark variance (C), angle (B), and percentage of nodes in the hierarchical part (A).

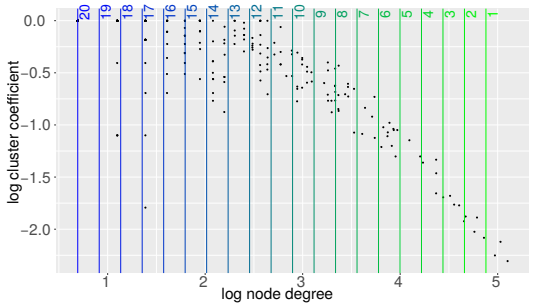

Fig. 5. The figure shows the hierarchy of one time slot of LLVM (2008-06 to 2008-12). Each vertical line denotes one possible split into a hierarchical part and a non-hierarchical part.

4.2.1 Manual Data Labeling. We used human annotation to obtain labels for 289 randomly selected developer networks from a subset of our subject projects (see Sec. 5.1). In this case, a labeled data point corresponds to a network and the position of a breakpoint for decomposing the network into hierarchical and non-hierarchical parts. The division into a hierarchical and non-hierarchical part was done in a consensus vote based on visual inspection by three of the authors, no specialized domain knowledge was required for this task. As no bias should be introduced into this judgement, no information on the project and its members was used during the process.

To perform this labeling task, human annotators were shown a figure with 20 candidate breakpoints (see Fig. 5) and asked to select one. We formed candidate groups by dividing the range of node degrees into 20 buckets (intervals) of equal length $b_1..b_{20}$. This choice provided a reasonable compromise between granularity and computational cost. The first candidate group for the hierarchical part contained the developers in the bucket with the highest node degrees (b_1), with a minimum of two developers. The second candidate group contained developers of the two buckets with the highest node degrees ($b_1 \cup b_2$), and the developers of the remaining 18 buckets ($\bigcup_{i=3..20} b_i$) for the non-hierarchical part. This continued until all developers are included in the candidate group for the hierarchical part, resulting in 20 candidate groups in total. The annotation instructions indicated that the breakpoint should be chosen such that points to the right exhibit a linear relationship with a negative slope and points to the left do not. Each network to be labeled was shown to multiple annotators and an average was taken to reduce error. For the example in Fig. 5, the annotators chose the buckets b_{10}, b_{11}, and b_{13}.

We used 80% (231) of the labeled examples for fitting parameters in our automated method, described in detail below. The remaining 20% (58) of the labeled examples were used to test the agreement between our automated method and human annotators on networks not used during parameter fitting. The results indicate that the automated method and human annotation differ on average by 10% (i.e., an error of two buckets). For the example in Fig. 5, our automated method (which we describe in what follows) selects b_{13}.

4.2.2 Combinatorial Optimization. The following describes our automated method for determining the hierarchical and non-hierarchical parts.

Step 1 (Compute metrics of the candidate groups). To choose the optimal split of the developers into a hierarchical and a non-hierarchical part, we start from 20 equally-sized *candidate groups*, which was a reasonable compromise between granularity and computational cost for our data set. Three metrics (also visualized in Fig. 4 as dashed line and arrows) for every candidate group served as input to an optimization procedure:

(A) the percentage of nodes of the hierarchical part (*perc*);
(B) the angle between regression line (red solid line in Fig. 4) and x axis (*angle*);
(C) and the residual variance of the regression fit (*var*).

The percentage of developers in the hierarchical part (*perc*) is defined as the number of developers (black dots) to the right of the dashed line divided by the number of all developers in the plot (all developers in the time range with two or more interactions). The solid line is the fitted regression line from a least-squares single linear regression of the log clustering coefficient on the log node degree. The *angle* between this line and the x axis (the bent arrow) is related to the hierarchical part, such that a higher angle indicates a stronger hierarchical structure [26, 39]. If this angle is zero, no hierarchy is present. The goal is to find the largest possible angle. We compute the residual variance (*var*) for the hierarchical part of the candidate group as the average squared deviation of the black dots right of the dashed line (illustrated by the double-sided arrows in Fig. 4). The regression line should be as close to each black dot (developer) as possible. Thus, the variance should be as low as possible. As result of Step 1, we know *perc*, *angle*, and *var* for each of the 20 candidate groups.

Step 2 (Apply thresholds to candidate groups). We aim at identifying the most suitable set of developers comprising the hierarchical part. To this end, the decision on whether to keep a candidate group or not is based on thresholds that we determined from the manual data labeling (as described above in Sec. 4.2.1), which we have performed on a sample of our subject projects, to filter degenerate cases (e.g., a large variance that blurs the hierarchical structure). Hence, based on all the selected buckets that resulted from the manual data labeling, we computed the following thresholds, which we now apply to all candidate groups:

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

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

Step 3 (Selection of optimal candidate group). We select one of the candidate groups $s \in S_t$ per time range that best describes the hierarchical and the (remaining) non-hierarchical part. To this end, we compute $\overline{S} = \bigcup_{t \in 1..T} S_t$, the union of the sets of candidate groups of all time ranges per project. As the range of the three criteria varies from project to project, we first standardized the values from *Step 2* project-wise by subtracting the project-specific means of the criteria (Equ. (1))

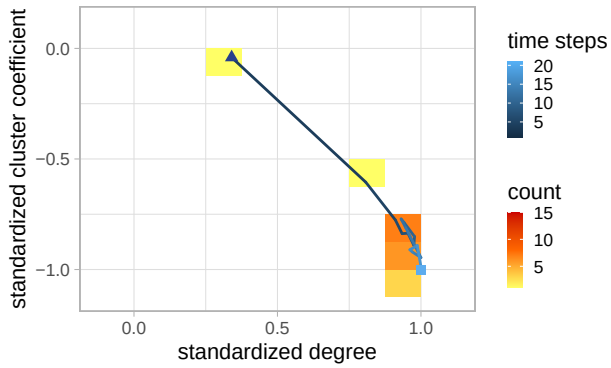

Fig. 6. The considered developer of the project Node.js (developer 874) starts in the non-hierarchical part (dark blue triangle) with a low number of contacts and a high clustering coefficient. Later, the developer moves to the hierarchical part, having many contacts and a low clustering coefficient, until the developer leaves the project (light blue rectangle). The color of the big rectangles in the background denotes the count of how many time ranges the developer has held a certain position; the darker the rectangle, the longer the developer has held the position.

and by dividing by the standard deviation of the criteria, (Equ. (2)). If m_s represents any of the three measures for candidate group $s \in \overline{S}$ in time range $t \in 1, \ldots, T$, and $N = |\overline{S}|$ is the number of all candidate groups for one project across all time ranges, the calculus is:

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

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

$$m_s^{std} = \frac{m_s - \overline{m}}{sd}. \tag{3}$$

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

$$\arg\max_{s \in S_t} \; (3 \cdot perc_s^{std} + angle_s^{std} - var_s^{std}) \tag{4}$$

This way, our method favors high values for criteria (A) and (B), but penalizes high values for criterion (C). This procedure results in one split per time range and project. Algorithm 1 summarizes the described steps. More details and examples are available on the supplementary website.

4.3 Change of Position in Hierarchy (RQ_2)

To address RQ_2, we track how a developer's position in the organizational structure changes over time. For this purpose, we normalize the clustering coefficient and node degree to analyze the position independent of the network size (i.e., number of developers) over the entire project history.

Algorithm 1: Divide developers into a hierarchical and a non-hierarchical part

Data: networks net_t for each time range $t \in 1..T$
Result: developers s_t of the hierarchical part
for $t \in 1..T$ **do**
  foreach developer i in net_t **do** ▷ *compute basic statistics*
    $k_i :=$ node degree of i
    $n_i :=$ number of links between neighbors of i
    $c_i := 2 \cdot n_i/(k_i \cdot (k_i - 1))$ ▷ *clustering coefficient*
  end
  divide range of $\log(k_i)$ into 20 buckets $b_1..b_{20}$
  $s := b_1$, $S_t := \{s\}$
  for $z \in 2..20$ **do** ▷ *compute candidate groups S_t for hierarchy*
    $s := s \cup b_z$, $S_t := S_t \cup \{s\}$
  end
  for $s \in S_t$ **do** ▷ *Step 1*
    compute $perc_s$, $angle_s$, and var_s
    if $\neg(perc_s > 0.05 \wedge angle_s > 35° \wedge var_s < 0.5)$ **then**
      $S_t := S_t \setminus \{s\}$ ▷ *Step 2*
    end
  end
end
$\overline{S} := \bigcup_{t \in 1..T} S_t$ ▷ *Step 3*
for $s \in \overline{S}$ **do**
  compute standardized values var_s^{std}, $perc_s^{std}$, $angle_s^{std}$
end
for $t \in 1..T$ **do** ▷ *selection of optimal candidate group s_t*
  $s_t = \arg\max_{s \in S_t} (3 \cdot perc_s^{std} + angle_s^{std} - var_s^{std})$
end
return $\{s_t \mid t \in 1..T\}$

The clustering coefficient ranges between $[0, 1]$, so its log ranges from $-\infty$ to 0. To track the position of individual developers in the hierarchy, we compare their position across different networks. To prevent distorting effects, we normalize the node degree and clustering coefficient across the networks of the project's history. We normalize the smallest clustering coefficient over a time range and project to a value of -1, by dividing all log clustering coefficients by the absolute value of the smallest log clustering coefficient. This way, the developer with the highest node degree and the lowest clustering coefficient of a time range is always on the same position in the hierarchy plot, irrespective of the network size. The closer the *normalized clustering coefficient* is to -1, the more important is the respective developer's role as hub. The log node degree is a positive number. We normalize it to a value in $[0, 1]$ by dividing all log node degrees by the highest log node degree in the given time range. Thus, the developer with the highest node degree has a *normalized degree* of 1 (see Fig. 6).

We describe the developer's *change of position* as changes in the developer's normalized position in the organizational structure. In Fig. 6, we show such a change with a trajectory and a heat map for a developer of the project Node.js. The considered developer starts (dark blue triangle) with a low node degree and a high clustering coefficient. Then, the developer's node degree increases while the clustering coefficient decreases (movement to the hierarchical part), until the developer reaches a very high position (bottom right of the plot) in the project community, which is stable

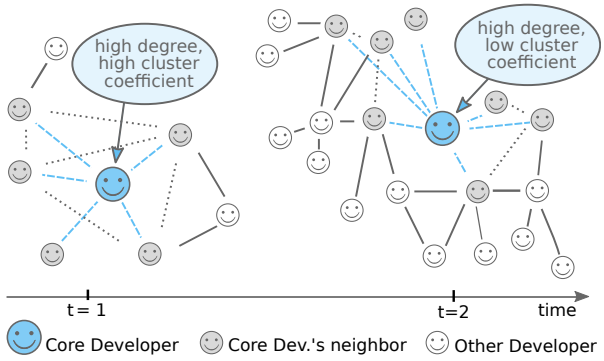

Fig. 7. Neighborhood of a selected developer (blue node) and corresponding neighbors (grey nodes) at two different time ranges.

for several time ranges. Thus, the developer has acquired many neighbors who do not interact a lot with each other, until the developer leaves the project (light blue rectangle).

We restrict our analysis to the 10 most active developers per project, as understanding the typical contribution and activity cycle of them allows us to analyze the on- and off-boarding process of socially active developers. To detect the 10 most active developers, we compute the total number of interaction partners per developer over the project history. To achieve a more complete perspective, we also select 10 developers randomly and analyze their changes of position. In addition, we also provide descriptive statistics for the selected developers (number of commits, number of e-mails or issue comments, respectively, the number of active time periods and how many of these time periods the developer was part of the hierarchical part), to give an impression of their activity.

4.4 Developers' Neighborhood (RQ_3)

For RQ_3, we extract the direct neighbors (i.e., first-order neighborhood) of the developers selected in RQ_2 and analyze the distribution of the neighbors regarding their position in the hierarchical or non-hierarchical part. Since the networks evolve with time, each neighborhood set is indexed by time as well (see Fig. 7). Based on how the networks are constructed, a neighbor is a developer who participated in, at least, one common communication activity (e-mail thread for mail networks or issue discussion for issue networks, respectively) within the six-months period on which the network has been constructed.

4.5 Tenure and Programming Activity (RQ_4)

To answer RQ_4, we compute tenure as the time between the first communication activity of a developer and the end date of the current time range for all developers. The end date of the time range represents the passed time for all developers, including the new developers in this time range. We compute the number of files edited by each developer within a time range and represent the results with scatter plots; we extract the number of edited files per developers from the version control system of the projects. For both, tenure and programming activity, we compare the developers of within and outside the hierarchy, globally as well as their trends over time. We use a one-tailed, unpaired Mann-Whitney U test for the global comparisons and report the corresponding p values as well as Cliff's Delta d, which quantifies the effect size corresponding to the statistical test.

Table 1. Overview of subject projects

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

[1] Date: time period of availability of mailing data (upper ten projects) and issue data (lower ten projects)
[2] # Global: number of active developers; # Max, # Min: maximal and minimal number of active developers (incl. developers with 1 or 0 contacts) in a time range; # First, # Last: number of active developers in first and last range
[3] kLOC: number of lines of code (LOC) in thousands, including comments and blank lines

5 LONGITUDINAL STUDY

To answer our research questions, we conducted an exploratory, longitudinal study on 20 popular and widely-used OSS projects.

5.1 Subject Projects

For our study, we selected 20 subject projects. Since OSS projects are of different age and use a wide array of communication channels [43], we focus on projects that either use a mailing list or an issue tracker as their main communication channel, from which we construct developer networks. Half of our subject projects uses a mailing list as their main communication channel, and the other half of them mainly uses GitHub issues; none of our subject projects uses both communication channels simultaneously. Both kinds of communication channels that we investigate, developer mailing lists and GitHub issues, are used for public, technical discussions among developers and for reviewing code changes (in terms of patches on mailing lists, in terms of pull requests on GitHub issues). Hence, both communication channels contain similar content and are used for similar purposes, which is why only either of the channels is used in each of our subject projects.

We cover projects from various domains (including compilers and virtualization software), programming languages (e.g., C, Python, JavaScript), and sizes (from 132 kLOC to 10 988 kLOC; see Table 1). To reduce bias due to noisy and incorrect data, we limited our selection of projects: For projects with publicly accessible mailing lists, we only select projects that have already been studied in the related literature [5, 25–27]. For the selection of projects that use the GitHub issue tracker, we considered the list of most starred GitHub projects in 2020.[7]

[7] https://www.attosol.com/top-50-projects-on-github-2020/

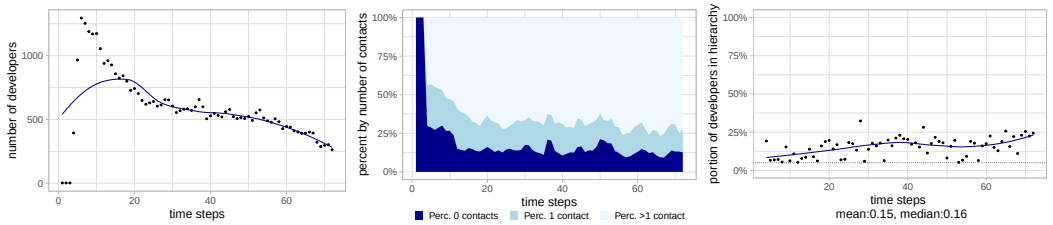

Fig. 8. Project GCC shows a typical evolution pattern: The number of developers increases, and after a certain period of time, the number decreases (left). The fraction of developers with no, one, or more than one contacts becomes stable (middle). The fraction of developers in the hierarchical part slowly increases (right).

5.2 Results

In this section, we summarize and discuss the most important results and observations of our study, and we derive a number of hypotheses to guide further work. As we generated a large amount of data, we present only selected cases and representative figures here (e.g., we use only results from the 6-month time ranges since they are most illustrative). The remaining data are available on the paper's supplementary website (SW). In what follows, we include links to the plots on the supplementary website in braces after the project's name. For example, (GiF) links to the plot of the programmer activity's analysis for project GIT of the form [abbreviation for project, (number of developer), abbreviation for analysis], directly leading to the specific plots.

Table 1 shows the maximum and minimum number of developers, as well as the number of developers at a project's start and end times. In all projects, the number of developers increases, reaches a peak, and then decreases. An exception is project WINE (WiA). There, we observe only a decrease. The typical behavior happens in four patterns: (1) a slow and steady increase is followed by a short decrease (ELECTRON (EA), LLVM (LA), QEMU (QeA), and REACT (RA)); (2) a steeper increase is followed by a slower and smooth decrease (with possible bumps), for ANGULAR (AnA), ATOM (AtA), DJANGO (DA), GCC (GcA), MOBY (MA), NODE.JS (NA), OWNCLOUD (OA), QT (QtA), and WEBPACK (WeA); (3) increase, decrease, increase: BOOTSTRAP (BA), FFMPEG (FfA), and GIT (GiA); (4) only increase: FLUTTER (FlA), TYPESCRIPT (TA), and U-BOOT (UA). An example for the most frequent pattern (2) is GCC (Fig. 8 (left)).

5.2.1 *Typical Structure and Evolution (RQ_1).*

Results. With an increasing number of developers, the proportion of developers in the hierarchical part decreases in most projects; and, with a decreasing number of developers, the proportion of developers in the hierarchical part increases (ANGULAR (AnA), ATOM (AtA), BOOTSTRAP (BA), ELECTRON (EA), FLUTTER (FlA), GCC (GcA), GIT (GiA), LLVM (LA), MOBY (MA), NODE.JS (NA), OWNCLOUD (OA), QT (QtA), QEMU (QeA), REACT (RA), TYPESCRIPT (TA), U-BOOT (UA), and WEBPACK (WeA)). We illustrate this in Fig. 8 (right) for GCC: The percentage of developers in the hierarchical part grows from 7% to 25%, while the developers' number falls from around 1 300 developers to around 260 developers. For the projects FFMPEG (FfA) and DJANGO (DA), the portion of the hierarchy and the number of developers is independent of each other. The portion of hierarchy of WINE (WiA) is mainly stable, but in the end, we find an increase of the portion.

Only in a low number of the analyzed time ranges, we do not find a hierarchical structure: ANGULAR (AnH) (range 1), ATOM (AtH) (range 1), FLUTTER (FlH) (ranges 16 and 17), GCC (GcH) (ranges 1, 2, and 3), GIT (GiH) (ranges 1 and 2), OWNCLOUD (OH) (range 31), QT (QtH) (ranges 1–6), U-BOOT (UH) (range 1), WEBPACK (WeH) (range 1), and WINE (WiH) (range 61). This mostly happens in the very

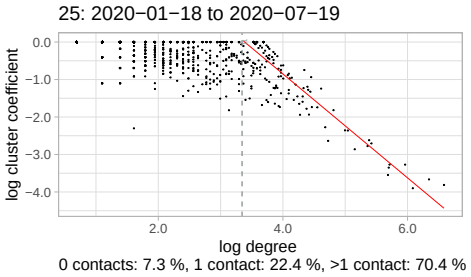

Fig. 9. Hierarchical structure of the 25th analyzed 6-month time range of project Angular: The plot shows, on a logarithmic scale, the node degree and clustering coefficient of all developers who participated in the issue discussions of this time range. We identified a hierarchical structure (to the right of the dashed line).

first time ranges of a project, where only very few developers are communicating yet, ending up in a loosely connected network where most developers have less than three connections to others.

Nonetheless, for almost all ranges of the above stated projects and all ranges for the projects Bootstrap (BH), Django (DH), Electron (EH), FFmpeg (FfH), LLVM (LH), Moby (MH), Node.js (NH), Qemu (QeH), React (RH), and TypeScript (TH), we are able to identify a hierarchical structure for all analyzed time ranges. To provide an example, we show the hierarchical part of project Angular in Fig. 9 (to the right of the dashed line). Over time, the residual variance of the regression fit and the angle between regression line and x axis (see Sec. 4.2.2) stay relatively stable for each project.

Answering RQ_1, we observe the presence of a hierarchical part for all analyzed projects. A pervasive phenomenon is that the proportion of developers in the hierarchy changes over time: At the beginning, when projects have few developers, almost all developers tend to be positioned within the hierarchical part; as the project matures and grows, the vast majority of developers exist outside of a hierarchy (up to 90%). Thus, we see evidence of temporal patterns that indicate a fundamental shift in the organizing principles at play in OSS projects. The fundamental shift is from hierarchy as a global organizing principle at an early project stage to a local hierarchical part with the vast majority not hierarchically organized in later project stages.

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

The presence of a hierarchical part tends to be unaffected by variations in the numbers of developers of a project; variation tends to be limited to fluctuations in its relative size as the project matures. Developers who enter or leave the hierarchical part change its composition. Despite this developer turnover, the slow change in residual variance and in the angle between the regression line (red solid line in Fig. 9) and the x-axis over time, as indicators for stable hierarchical structure, suggests that the subject projects have a stable organizational structure. This finding is in line with the hypothesis that the hierarchical part is principally responsible for coordination supporting information exchange [25]. In this case, one would indeed expect that successful projects achieve stable hierarchy, since large structural shifts disrupt coordination.

5.2.2 *Change of Position in Hierarchy (RQ_2).*

Results. In Fig. 10, we show how the position of an exemplary developer changes during the evolution of project LLVM. The developer starts in the upper left of the plot, that is, at the bottom of the hierarchy or already in the non-hierarchical part. Then, the clustering coefficient decreases as the developer moves down in the plot towards the hierarchical part, where the developer stays for around 12 time ranges. Then, the developer moves back to the non-hierarchical part and potentially leaves the project (if this is not yet the end of our analyzed time period).

We find notable patterns of positional change, which we summarize in Table 2 for mail networks and Table 3 for issue networks. In total, we analyze 200 developers per data source (i.e., 200 developers for issue networks and 200 developers for mail networks)—the 10 developers with maximum node degree and 10 random developers per project as described in Sec. 4.3. We provide descriptive statistics for both, the most active and the randomly selected developers, in Tables 6 and 7 in the appendix. In general, the majority of the randomly selected developers contributed no commit to the project and only few e-mails or issue comments, whereas the most active developers were not only highly active in communicating, but were also highly active code contributors. (Project GCC is an exception, where even none of the most active communicators contributed any commit to the source code; we discuss this phenomenon further below in Sec. 5.2.4.) As expected, the most active developers appear to be active in more time ranges than the randomly selected developers do. In line with that, the most active developers are mostly part of the hierarchical part, whereas the randomly selected developers are only rarely part of the hierarchical part.

The mail and issue networks exhibit largely similar movement patterns for the selected developers, so we focus on the details of mail networks to summarize. The movement patterns describe different starting points and directions of position changes in the organizational structure. The most frequent pattern occurs in all projects: 40 out of 100 most active developers start at the non-hierarchical part's upper left region in the hierarchy plot, then move down to the upper levels of the hierarchical part (lower right), to finally return to the non-hierarchical part again (pattern "down → up"). An example of this pattern is developer 1610 of the project LLVM (Fig. 10). The two second most frequent patterns (18 out of 100) describe developers who move from the non-hierarchical part to the upper levels of the hierarchical part (that is, down to the right in the hierarchy plot, pattern "down") and developers moving in the opposite direction (18 out of 100), that is, they start in the hierarchical part and then move to the non-hierarchical part (pattern "up"). The five remaining patterns play only a secondary role and do not occur often. In the end, we find 11 developers over all projects who have other (individual) movement patterns.

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

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

Answering RQ_2, we observed patterns of transitions regarding developers' trajectory through positions in the organizational structure. In the two most frequent patterns, developers move from the non-hierarchical to the hierarchical part. In one, they move back and in the other, they stay. Other transitions tend to be rare.

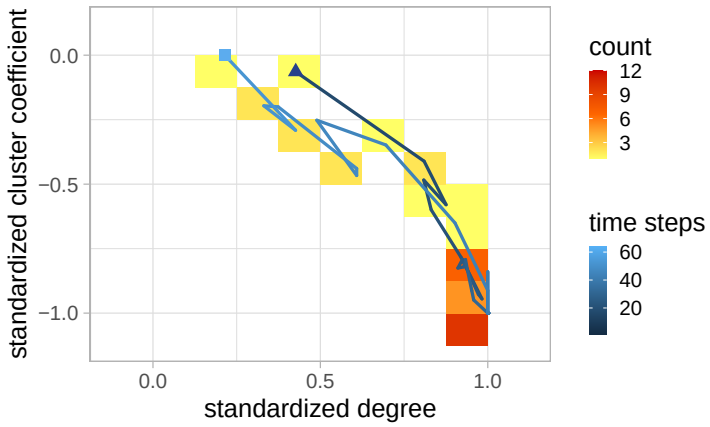

Fig. 10. The considered developer of the project LLVM (developer 1610) starts with a low number of contacts and a high clustering coefficient in the non-hierarchical part (dark blue triangle). The developer has many contacts in the middle of the project's analyzed time ranges, then they move to back to the non-hierarchical part, until they leave the project (light blue rectangle). Hence, the corresponding movement pattern for this developer is "down → up". The color of the big rectangles in the background denotes the count of how many time ranges the developer has held a certain position; the darker the rectangle, the longer the developer has held the position.

Table 2. Frequency of directions of positional change in the hierarchy for the 10 most active/10 randomly selected developers for mail networks.

Movement Pattern	Django	FFmpeg	GCC	git	LLVM	ownCloud[1]	QEMU	Qt	U-Boot	Wine	Σ	$\Sigma_{\text{all devs}}$
down → up	5/1	4/–	2/–	4/1	6/2	7/1	6/2	–/4	3/3	3/3	40/17	57
constant	–/5	1/3	–/5	1/2	–/4	–/–	–/6	2/3	–/5	1/2	5/35	40
down	2/1	1/–	1/–	2/–	2/1	1/3	3/–	1/–	5/–	–/1	18/6	24
up	1/–	3/–	1/–	1/–	–/–	1/2	1/–	6/1	1/–	3/2	18/5	23
horizontal	–/–	–/5	–/1	–/5	–/1	–/1	–/2	–/1	–/2	–/–	–/18	18
down → up → down	2/–	–/–	1/–	2/1	1/–	1/–	–/–	–/–	–/–	–/–	7/1	8
up → down	–/1	–/–	1/1	–/1	–/–	–/–	–/–	–/–	–/–	–/–	1/3	4
up → down → up	–/1	–/–	–/–	–/–	–/–	–/1	–/–	–/–	–/–	–/–	2/–	2
other	–/1	1/2	4/3	–/–	1/2	–/1	–/–	1/1	1/–	3/2	11/12	23

[1] There are only nine random selected developers in ownCloud.

Table 3. Frequency of directions of positional change in the hierarchy for the 10 most active/10 randomly selected developers for issue networks.

Movement Pattern	Angular	Atom	Bootstrap	Electron	Flutter	Moby	Node.js	React	TypeScript	webpack	Σ	$\Sigma_{\text{all devs}}$
down → up	2/1	7/–	4/–	4/–	1/–	7/–	3/–	4/1	3/–	6/–	41/2	43
constant	–/6	–/9	1/6	–/7	1/7	–/7	1/4	–/7	4/7	1/8	8/68	76
down	3/1	–/–	2/–	5/–	7/–	2/–	4/–	3/–	1/–	1/–	28/1	29
up	–/–	1/–	2/–	–/–	–/–	1/–	2/–	2/–	–/1	–/–	8/1	9
horizontal	–/1	–/1	–/3	–/2	–/2	–/2	–/3	–/2	–/2	–/–	–/18	18
down → up → down	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–
up → down	–/–	–/–	–/–	1/–	–/–	–/–	–/–	–/–	–/–	–/–	1/–	1
up → down → up	–/–	–/–	1/–	–/–	1/–	–/–	–/–	–/–	–/–	–/–	2/–	2
other	5/1	2/–	–/1	–/1	–/1	–/1	–/3	1/–	2/–	2/2	12/10	22

Discussion. The high number of occurrences for pattern "constant" for the randomly selected developers in the issue networks might be caused by the much higher number of project "users" participating in issue discussions only for a short time period, compared to the mail networks (see Table 1). When neglecting the pattern "constant" for the randomly chosen developers in issue networks, the most frequent pattern among all developers (most active and randomly chosen developers) is that developers enter the project with only few contacts (pattern "down → up"). Over time, the number of interaction partners rises and the developer climbs the project's hierarchy. This may be indicative of their role changing and gaining coordination responsibilities. Then, the number of interactions decreases again and the developer returns to a small number of contacts. The second most frequent pattern is similar to the first in that non-hierarchical developers move to the hierarchical structure's upper regions—however, either they stay or we run out of data before we see them leaving (pattern "down"). This might be caused by the much higher number of project "users" participating in issue discussions only for a short time period (see Table 1).

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

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

5.2.3 *Developers' Neighborhood (RQ_3).*

Results. We illustrate an example in Fig. 11 and summarize the results for all projects in Table 4 for mail networks and Table 5 for issue networks. First, we explore the most active developers in a project who have a static neighborhood. These developers interact during their entire life cycle with developers of the hierarchical part (2 out of 100 developers in mail networks, pattern "Hierarchical part") or both (61 out of 100 in mail networks, pattern "Both"). Second, the most active developers' neighborhood may change, too, which happens in two ways: Either a developer starts their career with contacts mainly from the hierarchical part, and then they interact with developers from both parts (17 out of 100 in mail networks, pattern "Hierarchical part → both"), or they start with contacts from both parts and then restrict their interaction to developers of the hierarchical part (19 out of 100 in mail networks, pattern "Both → hierarchical part"). Issue networks exhibit similar patterns as described for mail networks.

We also evaluate the neighborhood of randomly selected developers. Their neighborhoods are more stable. Those developers interact during their entire life cycle with developers of the hierarchical part (47 out of 99 in mail networks, pattern "Hierarchical part") or both parts (34 out of 99 in mail networks, pattern "Both"). Only 18 out of 99 randomly selected developers in mail networks have a dynamic neighborhood. For issue networks, similarly, only 10 out of 100 randomly selected developers have a dynamic neighborhood.

Answering RQ_3, the 10 most active developers interact (via e-mails or issues) with developers of the hierarchical part and sometimes, additionally, with developers of the non-hierarchical part, but rarely exclusively with developers of the non-hierarchical part. This also holds for the randomly selected developers, who mostly interact with developers of only the hierarchical part.

Discussion. If a project is assumed to have proficient leadership (which cannot be guaranteed for every project), then it is not unexpected that randomly selected developers of the non-hierarchical part mostly interact with developers of the hierarchical part or both groups, but not solely with developers of the non-hierarchical part: In any discussion, a developer of the hierarchical part

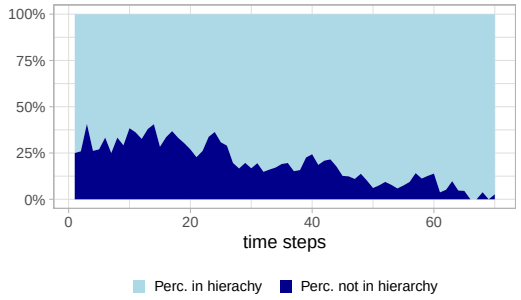

Fig. 11. Neighborhood of a developer (developer 7507) of the project U-Boot over time. That is, the percentage of the developer's contacts that are from the hierarchical (light blue) or from the non-hierarchical part (dark blue) at the respective time range.

Table 4. Frequency of position of neighborhood contacts in the hierarchy for each subject project for mail networks for 10 most active/10 randomly selected developers.

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

Table 5. Frequency of position of neighborhood contacts in the hierarchy for each subject project for issue networks for 10 most active/10 randomly selected developers.

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

can join to add clarifications or to make a decision, which is not unlikely given the role of the developers of the hierarchical part. Consequently, developers of the non-hierarchical part are expected to interact with developers of the hierarchical part in projects that have a well-functioning leadership. Also, it is worth mentioning that the developers of the non-hierarchical part do not deliberately choose their interaction partners, as they cannot influence who is replying to their messages. Developers from the hierarchical part, however, take the role of maintainers and, most likely, decide which discussions they reply to.

Especially the dynamic patterns of the most active developers' neighborhoods are interesting, as these shed light on on- and off-boarding processes. During on-boarding, developers start with interactions from the hierarchical part, and later extend their interaction to developers of both parts. This dynamic might suggest that, when developers enter a project, they start accumulating knowledge from developers of the hierarchical part and only later transfer knowledge to the non-hierarchical part. During off-boarding, we observe that developers focus their interaction to developers of the hierarchical part. Interaction with the non-hierarchical part or, more specifically, newcomers seems to be present, though. This finding suggests that, when central developers leave, they focus on bringing their ongoing topics to an end, but avoid opening new ones.

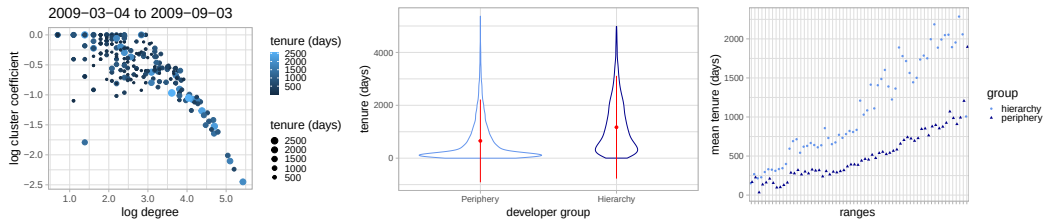

Fig. 12. Left: Tenure of the active developers of LLVM (03-2009 to 09-2009). The bigger and lighter a dot, the longer the developer is already active in the project; middle: distribution of tenure values per group; right: average tenure over time.

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

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

5.2.4 *Tenure and Programming Activity (RQ_4).*

Results. In Fig. 12, we show the developers' tenure and their position in the hierarchy for LLVM: The left plot encodes tenure in terms of size and color (larger and lighter dots denote shorter tenure values); the middle plot compares the distributions of tenure values of developers in the hierarchy with developers outside the hierarchy; the right plot shows the progression of average tenure values over time. Overall, the developers in LLVM's hierarchy have, on average, higher tenure values than the developers outside the hierarchy ($p \ll 0.001$; $d = 0.39$). This difference in tenure between developers inside and outside the hierarchy is consistent across all projects that use a mailing list ($p \ll 0.001$; $0.25 \leq d \leq 0.52$). Interestingly, the difference between tenure values of hierarchical and non-hierarchical developers increases over time. Remarkably, also this is consistent across all projects that use a mailing list, except for QT, where the difference stays constant over time.

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

Much like for tenure, we show the results for programming activity for developers of LLVM in Fig. 13. Developers in the hierarchical part edit most files ($p \ll 0.001$; $d = 0.40$). This difference in programming activity remains existent over time but is fluctuating with regard to its extent. For most projects, we find that, overall, the number of edited files of the non-hierarchical developers is significantly lower than the number of edited files of the hierarchical part. Only for GCC we cannot find any significant difference between developers inside and outside the hierarchy. As already seen for tenure, the difference between the number of edited files of developers inside and outside the hierarchy has a stronger effect on projects that use mailing lists (and project NODE.JS) ($p \ll 0.001$; $0.12 \leq d \leq 0.47$) than on projects that use the GitHub issue tracker ($p \ll 0.001$; $0.03 \leq d \leq 0.10$).

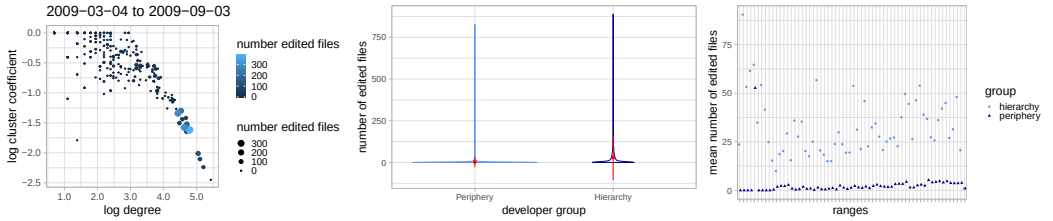

Fig. 13. Left: Number edited files of the active developers of LLVM (03-2009 to 09-2009). The bigger and lighter a dot, the more files the developer has edited; middle: distribution of the number of edited files per group; right: average number of edited files over time.

The dynamics of the individual projects show different patterns, though, which we group into four categories: In the first category, the upper hierarchical part contains both developers who edit many files and developers who edit no files or only a few files (Angular (AnF), Atom (AtF), Django (DF), FFmpeg (FfF), Flutter (FlF), git (GiF), LLVM (LF), ownCloud (OF), React (RF), TypeScript (TF), U-Boot (UF), and Wine (WiF)). In the second category, the distribution of the number of files edited is split between the hierarchical and the non-hierarchical part: the hierarchical part contains the developers who edit many files, whereas the non-hierarchical parts contains the developers who edit only few files (Bootstrap (BF), Electron (EF), Moby (MF), Node.js (NF), and webpack (WeF)). In the third category, the pattern is dynamic (QEMU (QeF) and Qt (QtF)). For example, in early phases of QEMU (QeF), mainly developers of the non-hierarchical part edited files. In later phases, most files were edited by developers of the hierarchical part. In the fourth category is only GCC (GcF), which has mainly editing developers in the non-hierarchical part of the network.

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

Answering RQ_4, we found a relationship between developers' tenure and their position in the hierarchy: Developers in the hierarchy have a longer tenure than developers outside the hierarchy. Regarding programming activity, developers of the hierarchical part perform most file edits. For some projects, this changes over time: In early project phases, mainly developers of the non-hierarchical part or both parts edit files; later, only developers of the hierarchical part edit files.

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

Hypothesis 4: Developers move quickly up the hierarchy when they take on coordination tasks early.

Hypothesis 5: Consistent contribution, coordination with other project members, and knowledge are important to role advancement in OSS projects.

The number of edited files seems to affect the position in the hierarchy more than the developer's tenure: the more files a developer edits, the more embedded they appear to be in the hierarchy, probably because a higher number of edited files increases the probability that their activity affects many other developers. The interesting cases are when the number of edited files and the position in the hierarchy are unrelated. This could be an indicator for a modular project structure, in which developers of the non-hierarchical part edit files of a certain part of the project, whereas the files that developers of the hierarchical part edit are scattered across many parts of the project. In project GCC, which is an outlier w.r.t. to the programming activity, there could also be an additional explanation for mainly having editing developers of the non-hierarchical part: As GCC is a rather low-level, technical project, which is dependent on technical features that rely on a certain hardware support [22], developers from different hardware manufacturers may add their specific hardware support to the code base, being in the non-hierarchical part of the project communication although accounting for many file edits. Table 6 in the appendix shows that none of the 10 most active developers on GCC's mailing list, who often are also in the hierarchical part, contributed any commit to the source code. This indicates that these most active developers on the mailing list take rather organizational coordination tasks than programming tasks in GCC, which is why this project has mainly editing developers in the non-hierarchical part of the mail network.

The relationship between developers' positions in the hierarchy and programming activity or tenure occurs to be less pronounced in issue-based projects than in projects that use mailing lists. One possible reason for that could be that there are lots of "users" of issue-based projects who participate in discussions for a large amount of time (e.g., reporting bugs, etc.), which also is represented in the sheer number of participants in the discussions (see Table 1). By contrast, there are many developers in the hierarchical part who perform merely project maintenance and pull-request reviews and therefore edit only a smaller number of files. Nevertheless, even if the effect is lower for issue-based projects, both kinds of projects have in common that the number of edited files and tenure are higher for developers inside the hierarchical part than outside the hierarchical part.

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

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

6 PERSPECTIVES

Overall, we found that the organizational structure co-evolves with OSS projects, despite the absence of external pressure to form any specific kind of structure. Specifically, it (a) regularly splits into a hierarchical and a non-hierarchical part, consistent with anecdotal evidence first seen by Joblin et al. [26]. Over time, the number of developers increases until reaching a peak, followed by a decrease, which is accompanied by a restructuring process. Growth in the number of developers usually leads to a decrease in the portion of developers who are organized hierarchically. From the perspective of individual developers, (b) the on-boarding of the most active developers follows typical patterns. We could confirm the belief that developers often start loosely connected in the non-hierarchical part [42], then tightly integrate into the hierarchical part, until they move to the

non-hierarchical part again and most likely leave the project. During this process, (c) developers' coding activity and their tenure can drive their hierarchical position. That is, on the one hand, early pioneers of a project are as likely to be in the upper ranges of the hierarchy as new members of a project. On the other hand, developer roles can adapt flexibly to changing project situations. This is relevant for research on success factors for climbing the social ladder of OSS projects [e.g., 17] and helps to better understand the social dynamics integrating project newcomers. Padhye et al. [37] found that core developers are open to accept bug fixes and documentation changes from peripheral developers, but not to proposed feature enhancements from peripheral developers. This habit appears inefficient—our method, when combined with a content analysis on the kind of contribution by the developers of the two parts, can help to identify such *community smells* [44].

Second, also at the group level, we observed considerable flexibility with regard to the number of groups in which developers want to engage. The prevalent hybrid structure that we observed contains developers whose clustering coefficient and node degree do not match the rules of hierarchical organization. Combining our approach with dynamic group analysis [e.g., 19] might provide insights into the hub function of central developers. For example, we observed that many former top developers leave the project with a high clustering coefficient, that is, their last interaction partners were from a well-connected subgroup. Further research on these groups shall complement analyses of developer tenure and knowledge conservation [e.g., 38]. Moreover, our method can be used to further investigate the role of developers within stable or dynamic subgroups of the project [e.g., 5], with respect to their position in the hierarchy to get more insights into how developers in the hierarchical part are connected to the different subgroups and to investigate their role with respect to programming activity and communication activity. This might provide further insights into group dynamics and may be used to derive recommendations for project managers on how to reduce developer turnover, knowledge loss, and maintain a project successfully. There is already evidence that socio-technical factors derived from network representations contain highly relevant information about the future success of a project [24]. The patterns found in our study are useful for providing additional context for predictive models to increase their effectiveness. Currently, predictive models lack the temporally rich patterns that we have identified. We believe that a coherent treatment of the temporal dimension is likely necessary to move beyond predictive analytics to achieve the end goal of prescriptive analytics.

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

7 THREATS TO VALIDITY

7.1 External Validity

We selected 20 OSS projects with a wide range of domains, programming languages, and sizes (see Table 1), so our results appear generalizable to similar projects. For generalization to different communication channels, we investigated projects that use mailing lists as their main communication channel as well as projects that use an issue tracker for communication, and we ended up in similar results for both types of projects. Clearly, 20 projects cannot cover the diversity of OSS projects out

there, and our study is naturally not meant to generalize to any OSS project. Still, some patterns and trends that we found are surprisingly pronounced and consistent across our subject projects. This demonstrates the principle power of our method to reveal such complex patterns and to study them in depth.

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

7.2 Construct Validity

The fact that we set the parameters of our classification algorithm only with a sample labeled manually threatens construct validity. To gain further confidence, we triangulated our classification with Joblin's dichotomous core–peripheral classification, which relies on the eigenvector centrality [26]. We found that the two classifications are consistent: In most projects, the set of core developers and developers in the hierarchy overlap to a large extent, but the core developers' number is often smaller than the number of developers in the hierarchy. Core developers are usually in the hierarchical part, but their percentage usually decreases as the project evolves.

A further threat to construct validity is the way we build our networks. As mailing lists or issue discussions, respectively, are the predominant communication channels in our subject projects, there was no need to consider further communication channels, such as Slack-chats or IRC. To learn about the relative influence of link types, we constructed for each network an additional version: one that consists of communication links and co-change links [27] (i.e., there is a link between a pair of developers if they have edited a common artifact within the respective time range). The rationale is that we get a more comprehensive view on the community if we include more information about the project. This also justifies the networks' construction and the combination of different link types (co-changes and communication). This way, we found that the developer networks we analyzed receive their structural properties mainly from the communication data (which are much more in number). The overall results of communication networks and the combined networks (co-changes and communication) are the same. We provide the results using these combined networks conveying communication and co-change information for all our research questions on our supplementary website (CombinedNetworks).

Another threat is the definition of tenure: We do not explicitly consider extended phases of developers' inactivity, during which developers do not contribute to the project. The tenure's end date is always the end of the time range, since we assume the developer's ongoing activeness. Our definition of programming activity poses a threat: Defining it as the number of edited files may be too simple, because the changes can be of different number of lines of code and different complexity. Counting the number of changes and giving weights to the ties accordingly would make our analysis much more complex, though. Zhou and Mockus [52] describe another method based on edited files that could improve a follow-up study.

8 CONCLUSION

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

their placement and trajectory through the hierarchy, and we analyzed the relationships of the developers' organizational position with their tenure and programming activity.

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

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

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

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (AP 206/14-1) as well as the Bavarian State Ministry of Education, Science, and the Arts in the framework of the Center Digitisation.Bavaria (ZD.B).

REFERENCES

[1] Bilal Afsar and Yuosre F. Badir. 2015. The Impacts of Person-Organisation Fit and Perceived Organisational Support on Innovative Work Behaviour: The Mediating Effects of Knowledge Sharing Behaviour. *International Journal of Information Systems and Change Management (IJISCM)* 7, 4 (2015), 263–285.

[2] Christian Bird, Alex Gourley, Premkumar T. Devanbu, Anand Swaminathan, and Greta Hsu. 2007. Open Borders? Immigration in Open Source Projects. In *Proc. Int. Workshop Mining Software Repositories (MSR)*. IEEE, 6–6.

[3] Christian Bird, Nachiappan Nagappan, Harald C. Gall, Brendan Murphy, and Premkumar T. Devanbu. 2009. Putting It All Together: Using Socio-technical Networks to Predict Failures. In *Proc. Int. Sympos. Software Reliability Engineering (ISSRE)*. IEEE, 109–119.

[4] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar T. Devanbu. 2008. Latent Social Structure in Open Source Projects. In *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 24–35.

[5] Thomas Bock, Angelika Schmid, and Sven Apel. 2022. Measuring and Modeling Group Dynamics in Open-Source Software Development: A Tensor Decomposition Approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 19:1–19:50.

[6] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. Who is Going to Mentor Newcomers in Open Source Projects?. In *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 1–11.

[7] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. 2008. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2–11.

[8] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering (TSE)* 35, 6 (2009), 864–878.
[9] Gemma Catolino, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Filomena Ferrucci. 2019. Gender Diversity and Women in Software Teams: How Do They Affect Community Smells?. In *Proc. Int. Conf. Software Engineering: Software Engineering in Society (ICSE-SEIS)*. ACM, 11–20.
[10] Can Cheng, Bing Li, Zeng-Yang Li, Yu-Qi Zhao, and Feng-Ling Liao. 2017. Developer Role Evolution in Open Source Software Ecosystem: An Explanatory Study on GNOME. *Journal of Computer Science and Technology (JCST)* 32, 2 (2017), 396–414.
[11] Kevin Crowston and James Howison. 2006. Hierarchy and Centralization in Free and Open Source Software Team Communications. *Knowledge, Technology & Policy* 18, 4 (2006), 65–85.
[12] Mariam El Mezouar, Feng Zhang, and Ying Zou. 2019. An Empirical Study on the Teams Structures in Social Coding using GitHub Projects. *Empirical Software Engineering* 24, 6 (2019), 3790–3823.
[13] Paul Erdős and Alfréd Rényi. 1959. On Random Graphs. *Publicationes Mathematicae* 6 (1959), 290–297.
[14] Michelle Girvan and Mark E. J. Newman. 2002. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826.
[15] Mehdi Golzadeh, Alexandre Decan, Damien Legay, and Tom Mens. 2021. A Ground-Truth Dataset and Classification Model for Detecting Bots in GitHub Issue and PR Comments. *Journal of Systems and Software (JSS)* 175 (2021), 110911.
[16] Mehdi Golzadeh, Damien Legay, Alexandre Decan, and Tom Mens. 2020. Bot or Not? Detecting Bots in GitHub Pull Request Activity Based on Comment Similarity. In *Proc. Int. Conf. Software Engineering Workshops (ICSEW)*. ACM, 31–35.
[17] Marvin Hanisch, Carolin Haeussler, Stefan Berreiter, and Sven Apel. 2018. Developers' Progression from Periphery to Core in the Linux Kernel Development Project. *Academy of Management Proceedings* 2018, 1 (2018), 14263.
[18] Andrea Hemetsberger and Christian Reinhardt. 2009. Collective Development in Open-Source Communities: An Activity Theoretical Perspective on Successful Online Collaboration. *Organization Studies* 30, 9 (2009), 987–1008.
[19] Steffen Herbold, Aynur Amirfallah, Fabian Trautsch, and Jens Grabowski. 2021. A Systematic Mapping Study of Developer Social Network Research. *Journal of Systems and Software (JSS)* 171 (2021), 110802.
[20] James D. Herbsleb, Audris Mockus, and Jeffrey A. Roberts. 2006. Collaboration in Software Engineering Projects: A Theory of Coordination. In *Proc. Int. Conf. Information Systems (ICIS)*. Association for Information Systems, 553–568.
[21] Pamela J. Hinds and Cathleen McGrath. 2006. Structures that Work: Social Structure, Work Structure and Coordination Ease in Geographically Distributed Teams. In *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW) & Social Computing*. ACM, 343–352.
[22] Bruno C. Honorio, João P. L. De Carvalho, and Alexandro J. Baldassin. 2018. On the Efficiency of Transactional Code Generation: A GCC Case Study. In *Workshop of Computer Systems and High Performance (WSCAD)*. IEEE, 184–190.
[23] Claus Hunsen, Janet Siegmund, and Sven Apel. 2020. On the Fulfillment of Coordination Requirements in Open-Source Software Projects: An Exploratory Study. *Empirical Software Engineering* 25, 6 (2020), 4379–4426.
[24] Mitchell Joblin and Sven Apel. 2022. How Do Successful and Failed Projects Differ? A Socio-Technical Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 67:1–67:24.
[25] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. 2017. Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 164–174.
[26] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. 2017. Evolutionary Trends of Developer Coordination: A Network Approach. *Empirical Software Engineering* 22, 4 (2017), 2050–2094.
[27] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. 2015. From Developer Networks to Verified Communities: A Fine-Grained Approach. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 563–573.
[28] Yuan Long and Keng Siau. 2007. Social Network Structures in Open Source Software Development Teams. *Journal of Database Management (JDM)* 18, 2 (2007), 25–40.
[29] Luis López-Fernández, Gregorio Robles, and Jesús M. González-Barahona. 2004. Applying Social Network Analysis to the Information in CVS Repositories. In *Proc. Int. Workshop Mining Software Repositories (MSR)*. IET, 101–105.
[30] Wolfgang Mauerer, Mitchell Joblin, Damian A. Tamburri, Carlos Paradis, Rick Kazman, and Sven Apel. 2022. In Search of Socio-Technical Congruence: A Large-Scale Longitudinal Study. *IEEE Transactions on Software Engineering (TSE)* 48, 8 (2022), 3159–3184.
[31] Andrew Meneely and Laurie Williams. 2011. Socio-technical Developer Networks: Should We Trust Our Measurements?. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 281–290.
[32] Andrew Meneely, Laurie Williams, Will Snipes, and Jason A. Osborne. 2008. Predicting Failures with Developer Networks and Social Network Analysis. In *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 13–23.
[33] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 3 (2002), 309–346.

[34] Nachiappan Nagappan, Brendan Murphy, and Victor R. Basili. 2008. The Influence of Organizational Structure on Software Quality. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 521–530.
[35] Mark E. J. Newman. 2005. Power Laws, Pareto Distributions and Zipf's Law. *Contemporary Physics* 46, 5 (2005), 323–351.
[36] Siobhán O'Mahony and Fabrizio Ferraro. 2007. The Emergence of Governance in an Open Source Community. *Academy of Management Journal* 50, 5 (2007), 1079–1106.
[37] Rohan Padhye, Senthil Mani, and Vibha S. Sinha. 2014. A Study of External Community Contribution to Open-source Projects on GitHub. In *Proc. Int. Workshop Mining Software Repositories (MSR)*. ACM, 332–335.
[38] Mehvish Rashid, Paul M. Clarke, and Rory V. O'Connor. 2019. A Systematic Examination of Knowledge Loss in Open Source Software Projects. *International Journal of Information Management (IJIM)* 46 (2019), 104–123.
[39] Erzsébet Ravasz and Albert-László Barabási. 2003. Hierarchical Organization in Complex Networks. *Physical Review E* 67, 2 (2003), 026112.
[40] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. 2009. Evolution of the Core Team of Developers in Libre Software Projects. In *Proc. Int. Workshop Mining Software Repositories (MSR)*. IEEE, 167–170.
[41] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering (TSE)* 37, 6 (2011), 772–787.
[42] Igor Steinmacher, Marco A. Graciotto Silva, and Marco A. Gerosa. 2014. Barriers Faced by Newcomers to Open Source Projects: A Systematic Review. In *Open Source Software: Mobile Open Source Technologies*. Springer, 153–163.
[43] Margaret-Anne Storey, Leif Singer, Fernando Figueira Filho, Alexey Zagalsky, and Daniel M. German. 2017. How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. *IEEE Transactions on Software Engineering (TSE)* 43, 2 (2017), 185–204.
[44] Damian A. Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. 2015. Social Debt in Software Engineering: Insights from Industry. *Journal of Internet Services and Applications (JISA)* 6, 10 (2015), 1–17.
[45] Damian A. Tamburri, Patricia Lago, and Hans van Vliet. 2013. Organizational Social Structures for Software Engineering. *ACM Computing Surveys* 46, 1 (2013), 3:1–3:35.
[46] Damian A. Tamburri, Fabio Palomba, and Rick Kazman. 2019. Exploring Community Smells in Open-Source: An Automated Approach. *IEEE Transactions on Software Engineering (TSE)* 47, 3 (2019), 630–652.
[47] Michael Weiss. 2018. Business of Open Source: A Case Study of Integrating Existing Patterns Through Narratives. In *Proc. Europ. Conf. Pattern Languages of Programming (EuroPLoP)*. ACM, 23:1–23:4.
[48] Mairieli Wessel, Bruno M. de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana P. Chaves, and Marco A. Gerosa. 2018. The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Proceedings of the ACM on Human-Computer Interaction (HCI)* 2, CSCW (2018), 1–19.
[49] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting Build Failures Using Social Network Analysis on Developer Communication. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 1–11.
[50] Yunwen Ye and Kouichi Kishida. 2003. Toward an Understanding of the Motivation of Open Source Software Developers. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 419–429.
[51] Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. 2017. On the Scalability of Linux Kernel Maintainers' Work. In *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 27–37.
[52] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 137–146.
[53] Minghui Zhou and Audris Mockus. 2011. Does the Initial Environment Impact the Future of Developers?. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 271–280.
[54] Minghui Zhou and Audris Mockus. 2012. What Make Long Term Contributors: Willingness and Opportunity in OSS Community. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 518–528.
[55] Minghui Zhou and Audris Mockus. 2015. Who Will Stay in the FLOSS Community? Modeling Participant's Initial Behavior. *IEEE Transactions on Software Engineering (TSE)* 41, 1 (2015), 82–99.

A APPENDIX

Descriptive Statistics for the 10 Most Active and 10 Randomly Selected Developers

Table 6. Descriptive statistics for the 10 most active and 10 randomly selected developers: Commit count and event count. For each of the two groups of developers, we report the minimum value and the maximum value per project as well as the 25%, 50%, and 75% quantiles.

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

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

Table 7. Descriptive statistics for the 10 most active and 10 randomly selected developers: Number of active time periods (i.e., number of overlapping 6-month ranges within the date denoted in Table 1 in which the developer contributed to the mailing list or commented on an issue) and number of time periods in which the developer was part of the hierarchical part. For each of the two groups of developers, we report the minimum value and the maximum value per project as well as the 25%, 50%, and 75% quantiles.

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