International Journal of Software Engineering
and Knowledge Engineering
Vol. 31, No. 8 (2021) 1213–1234

DOI: 10.1142/S0218194021500388

World Scientific
www.worldscientific.com

Dual Channel Among Task and Contribution on OSS Communities: An Empirical Study

Yu Zhang*, Yue Yu†, Tao Wang‡, Zhixing Li§ and Xiaochuan Wang¶

National University of Defense Technology
Changsha 410073, P. R. China
**zhangyu_@nudt.edu.cn*
†*yuyue@nudt.edu.cn*
‡*taowang2005@nudt.edu.cn*
§*lizhixing15@nudt.edu.cn*
¶*mrwangxc@nudt.edu.cn*

Received 16 March 2021
Revised 31 March 2021
Accepted 23 May 2021

Open Source Software (OSS) community has attracted a large number of distributed developers to work together, e.g. reporting and discussing issues as well as submitting and reviewing code. OSS developers create links among development units (e.g. issues and pull requests in GitHub), share their opinions and promote the resolution of development units. Although previous work has examined the role of links in recommending high-priority tasks and reducing resource waste, the understanding of the actual usage of links in practice is still limited. To address the research gap, we conduct an empirical study based on the 5W1H model and data mining from five popular OSS projects on GitHub. We find that links originating from a PR are more common than the other three types of links, and links are more frequently created in Documentation. We also find that average duration between development units' create time in a link is half a year. We observed that link behaviors are very complex and the duration of link increases with the complexity of link structure. We also observe that the reasons of link are very different, especially in P–P and I–I. Finally, future works are discussed in conclusion.

Keywords: Open source software; issue; pull request; link; empirical study.

1. Introduction

The widespread use of Open Source Software (OSS) not only creates a paradigm of popular distributed software development model [1–4], but also creates a self-learning and self-organizing OSS community [5, 6]. With the continuously evolving of distributed software development, pull-based development has become a

† Corresponding author.

recent form and gained tremendous traction in the OSS community. Compared with traditional models of OSS development (e.g. mailing list and issue tracking system), the pull-based development model is more popular in code integrating [7] because it provides process automation and information centralization [8, 9]. This model decouples the development effort from the decision-making to integrate code. In pull-based development, contributors submit file changes [10, 11], report defect findings, require features and ask questions [12, 13], while integrators oversee the merge process, provide feedback, make decision [14] and maintain the sustainability of the project [15, 16].

Due to mutual dependence [17], a development unit (i.e. issue and PR) is often referenced by other development units which triggers the creation of links [18]. The usage of links in multiple development processes (e.g. committing, creating and discussing) [7, 19] stems from various motivations of practitioners and enhances the dependency of tasks and contributions in OSS communities [20]. Link also involves stakeholders' consideration of issue resolution [21], and improves the efficiency and effectiveness of development manifested by the increased acceptance of contribution and decreased resource waste in code review [8, 22]. In addition, the link facilitates knowledge sharing in OSS community [23] because it contains a large amount of knowledge from the OSS project [24, 5].

To make use of the link properly, it is necessary to examine it from multiple perspectives [25, 26]. Previous researches have focused on the applications of link in code review recommendation [27], similar bug recommendation [28] and related knowledge acquisition [23]. Meanwhile, researchers also explored the reasons why developers leave links, the context where link appears and the potential impact of link on software development [21]. However, little is known about the real usage of links in the perspectives of customs, patterns and characteristics. A better understanding of this can help OSS practitioners to use the linking mechanism in a formal way and provide insights into the design of current tools for better support and best practices on efficient collaboration.

To bridge this gap, we provided an empirical study on five popular OSS projects hosted on GitHub to explore the comprehensive usage of link in practice. We borrowed the idea of the 5W1H model [29], which is a popular model describing a fact using who, what, where, when and how questions, and raised five research questions. We organized the knowledge about links in GitHub, depicting the developer (who) create a link (what) in a specific artifact (where) at a certain time (when) while expressing link behavior (how) for a specific intention (why) [30]. In this paper, research questions do not include "Who" because users in OSS community are obviously integrators and contributors that we all consider carefully.

In our study, first, we collected all historical development units of five projects via GitHub official API.[a] Second, we identified links from collected development units.

[a] https://docs.github.com/en/graphql.

Based on a total of 246,569 links, we investigated the real usage of link in multiple aspects to address the following research questions:

*RQ*1: *What are the types of link?*

In this research question, we first defined link types according to the type of source and target development unit and then analyzed distributions of link types.

*RQ*2: *Where do links appear?*

With this research question, we aimed to explore specific locations where links are used. We also studied the frequency of different locations.

*RQ*3: *When do links happen?*

In this research question, we first defined two time-related measurements which are *create time interval* (CTI) and *link time interval* (LTI) and then measured distributions of links on these measurements.

*RQ*4: *How are links organized?*

We first reorganized a set of connected links into new structures (i.e. multi-target link and cluster). Then we studied the complexity of new structures by analyzing link distributions on several metrics.

*RQ*5: *Why are links used?*

With this research question, we conducted a qualitative analysis to reveal reasons of link and presented the distribution according to the category of link type.

The contributions of this paper can be summarized as follows:

- We conducted a comprehensive and systematic empirical study on the practical usage of link in GitHub. We found that linked development units are close in time and obvious in location while composing relatively small size structures and resulted by various reasons.
- We provide actionable suggestions and implications for OSS practitioners and tool designers, which is useful in merging external contributions more effectively, maintaining project awareness and devising automatic tools.

The remainder of the paper is organized as follows. Section 2 describes the background of this paper. Section 3 presents the construction of the dataset used in this study. Section 4 reports the results and findings. Finally, Sec. 5 concludes the paper and outlines directions for future work.

2. Background

2.1. *Challenges in pull-based development*

Some preliminary researches have focused on the use of pull-based development [31–33, 19] and revealed that it provides fast turnaround, increased opportunities for community engagement and decreased time to incorporate contribution [8] which make it popular in distributed collaboration of OSS development [34, 35]. However, in practice, the challenges imposed by pull-based development [36] cannot be ignored.

Gousios *et al.* [37] found that integrators struggled to maintain the quality of their projects and had difficulties with prioritizing contributions [9, 38]. Popular projects

receive a big volume of contributions each day that burdens integrators for it is difficult to manage and decide the priority with the weak relationships among contributions. Although contributors had a strong interest in maintaining awareness of project status [39], it is also a challenge that contributors work in the way that there is a shortage of centralized coordination and organized development units [40]. For instance, several researches have pointed out that duplicate pull request is one of the reasons that hinders developers to contribute [41–43]. The undiscovered and not managed links among development units result in duplicate pull requests directly in contributors' work without realizing others' work. In addition, as for newcomers in the OSS community, Balali *et al.* [44] found 44 barriers newcomers faced. The barrier *lacking newcomer's background knowledge* calls for making full use of links among development units to formulate necessary knowledge and awareness of projects.

2.2. *Link in development units*

Comparing with traditional software engineering which relies on employment contracts, formal policy and hierarchical management to implement decision-making [15], OSS projects promote interest-led software development and attract voluntary work from contributors. Aberdour [15] listed differences between traditional software and OSS development in 11 perspectives, such as *Team members are assigned work* versus *Team members choose work* and *Much effort put into project planning and scheduling* versus *Little project planning or scheduling.* Knowledge management plays a significant role in both traditional software and OSS for the knowledge-intensive native of software development [45]. Document management used to manage explicit knowledge in artifacts of software and competence management used to organize tacit knowledge embedded within developers are fundamentals for knowledge management in traditional software engineering [46]. In OSS projects, benefiting from the maturity of OSS platforms and communities, explicit knowledge is well organized in files, versions and repositories. However, tacit knowledge management has not been studied in depth.

Link is a key component in supporting contributions and knowledge management in OSS within the pull-based development model [18] for it maintains awareness of the project and organizes knowledge embedded in various practitioners [18]. Figure 1 shows an example of a link in GitHub. In issue *rails/rails#40534*, contributor *ojab* left a comment and referenced to PR *rails/rails#40390* and PR *rails/rails#40385.* The two links triggered timeline events in corresponding PRs as shown in the right of Fig. 1.

In our study, the definition of link is informed as follows:

> *Link is a pair of development units from the source to the target. The source is the development unit in which the practitioner uses a link and the target is the development unit referenced by the practitioner.*

Due to the increasing importance, link in development unit has attracted attention from researchers interested in online collaboration and software development

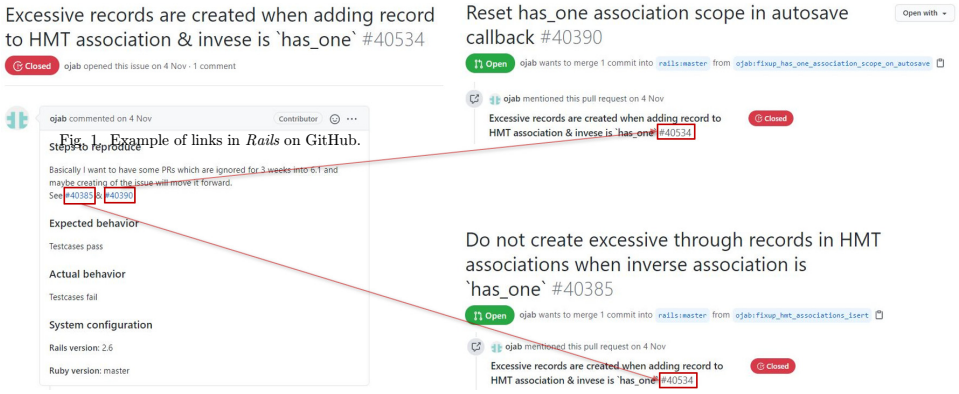


Fig. 1. Example of links in *Rails* on GitHub.

practices. According to Xu *et al.* [47], in Stack Overflow, they figured that knowledge units were linkable for different purposes and implemented a deep-learning approach to recognize different classes of linkable knowledge. Zhang *et al.* [17] presented a mixed-method study of issue linking and issue resolution in *Rails* on GitHub, in which they reported that developers tended to link more cross-project and cross-ecosystem issues which were associated with more discussions. Yu *et al.* constructed a large dataset of historical duplicate PRs in GitHub [42], which was one type of link, and analyzed redundancy, context and preference of duplicate PRs [24]. In addition, researchers have devoted themselves to tool implementation. Zhang *et al.* [23] proposed *iLinker* as a novel approach to accomplish recommendation tasks in acquiring related issue knowledge. Rocha *et al.* [28] implemented the related bugs recommendation tool *NextBug* for contributors and evaluated the tool to clarify the applicability, benefits and limitations of similar bugs recommendations.

3. Dataset

In this section, we present how the data was collected and processed. First, we selected five popular OSS projects hosted on GitHub, as described in Sec. 3.1. Second, we collected all historical development units through GitHub API in Sec. 3.2. Finally, we presented the more elaborate processing steps on link extraction in Sec. 3.3.

3.1. *Studied projects*

To conduct our study, we selected five projects from GitHub, as shown in Table 1. The criteria of project selection are that (i) they should have full-fledged and heavy usage of pull-based development model (e.g. all have more than 30,000 development units), (ii) they should be relatively popular and receive plenty of attention from the community, which can be quantified through the number of stars and forks (41.2 k stars and 16.0 k forks on average) and (iii) they should cover different

Table 1. Overview of studied projects.

Project	Language	Application	#PR	#Issue	#Star	#Fork
Elasticsearch	Java	Search engine	40.8 k	26.0 k	53.2 k	19.1 k
Joomla-cms	PHP	Content management system	20.0 k	11.6 k	3.7 k	3.1 k
Kubernetes	Go	Container management	60.3 k	37.1 k	73.6 k	26.8 k
Pandas	Python	Data analysis and manipulation	19.0 k	19.5 k	28.1 k	11.8 k
Rails	Ruby	Web framework	26.4 k	14.3 k	47.4 k	19.1 k

programming languages and application domains which increases the generalizability of our study.

3.2. *Data collection*

We used the GitHub GraphQL API to download historical development units along with their corresponding public release of selected projects. GraphQL API provides a new conceptual framework for implementing the way of data access interface with graph query language [48]. We followed the following steps to collect the data:

(i) For each project, we obtained historical development units by querying objects `issues` and `pullRequests` (referred as `developmentUnits`) as well as complementary information, such as `number`, `author`, `title`, `body`, `url` and `createdAt` in `developmentUnit`. We got 274,777 development units in total.
(ii) In each `developmentUnit`, we crawled object `comments` for the list of comments along with the corresponding segments (i.e. `author`, `body` and `createdAt`) and got 1,823,883 comments.
(iii) We also collected `CrossReferencedEvents` and `ReferencedEvents` from object `timelineItems` in each `developmentUnit`. They returned events triggered by link behaviors from other development units to the studied one. Meanwhile, segments, such as `actor`, `createdAt`, `subject`, `source` and `target`, were collected together. The number of collected referenced events and cross referenced events we got are 342,948 and 8,657,124, respectively.

3.3. *Link extraction*

We focused link extraction on the integrity of links in studied projects. First, we used data from `timelineItems` to extract explicit links as described in Sec. 3.3.1. Second, we analyzed content in `title` and `body` to extract implicit links in Sec. 3.3.2. Currently, we narrowed down the source and target development units of a link to the same project.

3.3.1. *Extracting explicit links*

In this part, we parsed event lists returned from `CrossReferencedEvent` and `ReferencedEvent`. Events in `CrossReferencedEvent` are triggered by link behaviors

when creating or discussing a development unit, while events in `ReferencedEvent` are triggered by link behaviors in commit message when committing changes of files. Each event includes information of actor, create time, source and target development units of the link, with which we converted events into links in our dataset. In total, we extracted 165,456 links from `CrossReferencedEvent` and 39,137 links from `ReferencedEvent`.

3.3.2. *Extracting implicit links*

To ensure the integrity of our dataset, we also extracted implicit links in `title` and `body` for several reasons. First, in early era of GitHub, *Cross Referenced Event* and *Referenced Event* were not applied to record links. For example, in comment of *rails/rails#371*, author *baroquebobcat* referenced development unit *rails/rails#451* ("*I put together a patch #451*"), but there is no event created in *rails/rails#451*. Second, links in development unit's title cannot be converted into events either. Thus, it is necessary to extract implicit links.

First of all, we extended a data preprocessing procedure to systematically clean raw data by eliminating unnecessary artifacts that affect implicit link extraction. The data preprocessing consists of the following steps: (i) eliminate *Block Quote* in comments which starts with a block marker ">" and brings a considerable number of duplicate links to the dataset, (ii) eliminate both *Code Block* and *Code Span* to remove invalid links from our dataset. Code block is marked with three consecutive backtick characters, while code span is marked with a backtick character. We applied the Python RegEx to do the data preprocessing and the resulting contents were prepared for extraction process elaborated as follows:

Extracting links in shortened format. According to GitHub Flavored Markdown [49], a link is automatically created when an author uses shortened format of text. The format *User/Project#Num* [50] (e.g. *rails/rails#26*) creates a link to the development unit numbered as *Num* in *User/Project*. Meanwhile, patterns of *#Num* (e.g. *#26*) and *GH-Num* (e.g. *GH-26*) create a link to the development unit numbered as *Num* in the same project. We used regular expressions to extract link in these patterns.

Extracting links via URL. URLs usually appear in plain text of raw data crawled from GitHub API to represent links. For example, "*https://github.com/rails/rails/pull/26*" creates a link to the development unit whose tracking number is 26 in *Rails* on GitHub. We also extracted all links in the format of URL using regular expressions.

The number of implicit links we got is 41,976.

It is worth noting that when extracting, we processed two types of link in a particular way: (i) *Replicate links*. When two development units are linked more than once, they compose a number of replicate links. We treated replicate links as the same one and only kept the earliest one to maintain the validity of dataset.

(ii) *Self-link.* If a development unit is referenced by itself, it creates a self-link. We removed all self-links for it is meaningless in our study. Overall, the details about the number of links in each project are shown in Table 2.

Table 2. Overview of the number of links.

Project	Elasticsearch	Joomla-cms	Kubernetes	Pandas	Rails	Total
#Links	58,137	25,985	110,928	32,211	19,308	246,569

4. Experiments and Results

In this section, we conducted experiments on 246,569 links and answered research questions of this paper. The research methods and results are illustrated as follows.

4.1. *RQ*1: *What are the types of link?*

With this research question, we first defined link type and then explored the distributions of links on different types.

4.1.1. *Link types*

In GitHub, issue and PR share the same numbering system. Therefore, in addition to referencing an issue/PR in an issue/PR, developers can also reference an issue/PR in a PR/issue. As shown in Fig. 2, links in GitHub can be classified into four types according to the type of source development unit and target development unit.

(i) P–P: A link from PR to PR,
(ii) P–I: A link from PR to issue,
(iii) I–P: A link from issue to PR,
(iv) I–I: A link from issue to issue.

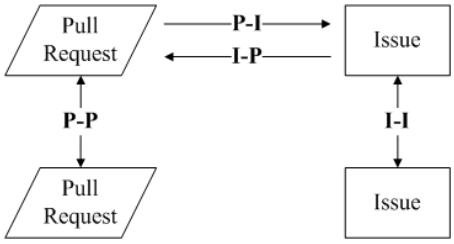


Fig. 2. Links among pull requests and issues in a project.

4.1.2. *Type distribution*

We present the distributions of links on four types in Fig. 3. We observed that 37.29% links start from issue, while 62.71% links are from PR, which is twice as much

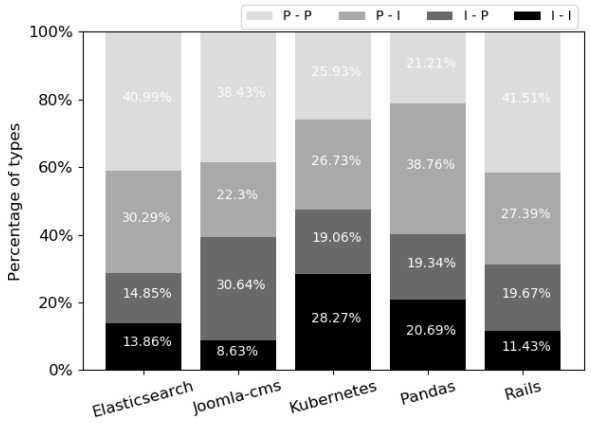


Fig. 3. Percentage of link types across studied projects.

as links from issue. It suggests that development behaviors in PRs are more complicated and attract more attention from practitioners in community. Links in PRs are worth spending more time discussing and reviewing. Especially, when solving issues, it is helpful to examine overall links associated with the development unit in order to avoid resources waste and mutual interference. In the perspective of target development unit, 54.33% links are directing to PR and 45.67% links are directing to issue. The relatively balanced distribution indicates that PR and issue are equally important artifacts and both contain development knowledge of OSS project. When figuring links, we should treat them on an equal footing. Among four link types, links in P–I and I–P are more closely related to issue resolution for they connect task and contribution directly. However, in the distributions, these links account for only half (49.81%) of the total. It means that contributors and integrators have to spend at least half effort in solving occasional problems other than issue resolution directly.

Summary. *Both the source and target of a link may be either an issue or a PR. Links originating from a PR are more common than the other three types of links.*
Implication. *When developers are submitting or discussing a PR, they should pay more attention to collecting context information since PR has higher chances of occupying relationships with other issues/PRs. For automatic tools identifying potential links, it is reasonable to consider the diversity of links and dedicate more effort to recommending links for PRs.*

4.2. *RQ2: Where do links appear?*

In this research question, we investigated the location of a link and analyzed the distributions of links in the perspective of locations.

4.2.1. *Link locations*

In GitHub, development unit is made up of multiple artifacts. Practitioners are allowed to use link in several artifacts. Thus, we defined the location of a link as the artifact where the link was used as follows:

- *Documentation*: The location includes title and body of a development unit. Links in documentation are already detected by contributors at the beginning of the development unit.
- *Comment*: Links in comment are created when discussing around the development unit. Compared with links in documentation, they are more time-consuming and difficult to detect as well as involving more resources and knowledge as the discussion continues.
- *Commit*: When committing changed files, contributors write down the commit message to describe the change briefly, in which a link would be created.

4.2.2. *Location distribution*

Figure 4 shows the distributions of link locations. On average, half of the links (56.65%) are created in documentation, where contributor writes concise information of the work she/he is going to do. It indicates that they have discovered these links at the first time of the development unit. However, to maintain the awareness and conduct efficient development of project, the proportion is not enough. Development units without link in documentation are created with no explicit announcement of related tasks and contributions which is not suggested by integrators of project. The 39.72% links are created in comment on average and discovered by different contributors. As for links which have already existed at the beginning of development unit, it is a waste of time and resources to arouse unnecessary communication rounds involving a number of contributors. It is a better choice for contributor to examine

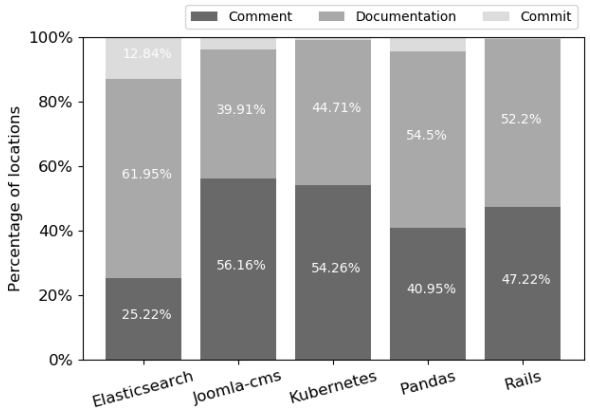


Fig. 4. Percentage of link locations across studied projects.

the overall links before contributing. As for links in commit, they account for only 9.63% on average. We infer the reason of scarcity of links in commit is that commit message is not displayed in the web page of development unit directly. It is to say that other contributors interested in the development unit cannot get the link information at the first glance of the development unit. It is not instrumental enough in exchanging contributors' opinions.

Summary. *We found that links may appear in three artifacts (i.e. Comment, Documentation and Commit) of a development unit. More than half of links in three projects (i.e. Elasticsearch, Pandas and Rails) are created at the submission of issue/PR (i.e. links appearing in issue/PR documentation).*
Implications. *We recommend that integrators should suggest in contributing guidelines that contributors search potential links before submitting an issue/PR and document such links, if any. For tool designers, they should detect related issues/PRs in not only issues/PR discussion but also issues/PR submission.*

4.3. *RQ3: When do links happen?*

The third research question elicits the time-related measurements. We first defined two statistical measurements of time factor, and then investigated the distributions of them.

4.3.1. *Time interval*

Time is an important metric in the analysis of link. For example, the longer a development unit lasts, the more likely it causes duplication [24] and the more links it may introduce. We defined two time statistical measurements:

(i) CTI: The time interval from the create time $timestamps_c$ of the source development unit d_s to the create time of the target development unit d_t in a link l:

$$\mathrm{CTI}(l) = timestamps_c(d_t) - timestamps_c(d_s).$$

(ii) LTI: The time interval from the latest create time of the source and target development unit to the link time $timestamps(l)$ of a link l:

$$\mathrm{LTI}(l) = timestamps(l) - \mathrm{Max}(timestamps_c(d_s), timestamps_c(d_t)).$$

As for CTI, the value is whether positive or negative. When CTI is negative, it indicates that links are recommending contributors to refer to historical development units, while to a latter development unit when the value is positive. The negative CTI links is more conspicuous than positive one for links of negative CTI are created in the latest development unit which is easier to discover. CTI is designed to reveal how long does it take to form a link. Another measurement LTI is definitely positive and shows how long does it take to discover a link after it is already formed.

We parsed GitHub API returned data `createdAt` in Sec. 3.2 to get $timestamps_c(d_s)$ and $timestamps_c(d_t)$. Next, we presented the following method we used to parse $timestamps(l)$ according to the location where the link l appears in Sec. 4.2:

(i) *Documentation*: The create time of link in documentation is as same as the create time of the development unit it belongs to. Thus, the $timestamps(l)$ equals the create time of source development unit $timestamps_c(d_s)$.
(ii) *Comment*: Link in comment is created along with the comment, so the $timestamps(l)$ is the create time of the comment.
(iii) *Commit*: As same as link in comment, $timestamps(l)$ of link in commit is the create time of the commit.

4.3.2. *Time interval distribution*

For better understanding of link behaviors in time dimension, we explored distributions of CTI and LTI. Figure 5 shows the results of this research question.

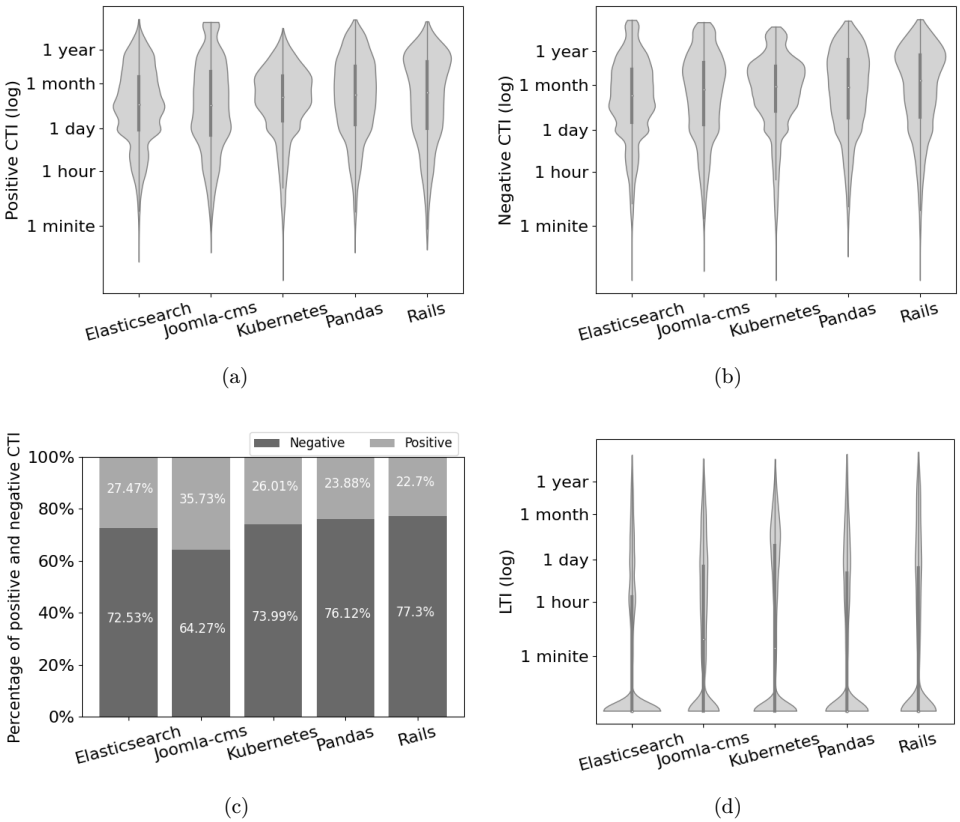


Fig. 5. Results of experiments on CTI and LTI: (a) statistics of positive CTI; (b) statistics of negative CTI; (c) percentage of positive and negative CTI and (d) statistics of LTI.

Figures 5(a) and 5(b) present the distributions of CTI in positive and negative, respectively. We found that the maximal duration of forming a link lasts extremely long, approaching the age of the project. On the other hand, the average CTI of all projects is 175.2 days, which has provided contributors a time window in which they should check related development units carefully. In Fig. 5(c), we observed that the number of links whose CTI are negative is significantly more than positive one (2.7 times on average). Account for the time sequence of development unit in issue tracker system and the difference of positive and negative CTI links mentioned above, we speculated that contributors are more inclined to create links in later development unit, which are more obvious in issue tracker system. We also observed that in Fig. 5(d), it takes 16.66 days on average to find a link after the formation of it. What is more, the longest LTI is 2788.6 days in *Pandas*. The lack of adequate link information leads to time waste in LTI which could be compressed largely with the help of automatic recommendation tools of link.

Summary. *The average value of CTI is about half a year, while the average LTI is half a month which cannot be ignored.*
Implications. *Integrators and contributors should pay more attention on issues/PRs created in past half a year when they are searching for potential related issues/PRs. As for tool designers, the automatic link detection model should consider the time factor to set a search time window, e.g. half a year. Moreover, the latency of link discovery raises the need for automatic tools to detect related issues/PRs in a timely manner.*

4.4. *RQ*4: *How are links organized?*

In this research question, we reorganized links into two new structures and analyzed their features, respectively.

4.4.1. *Multi-target link*

When analyzing links, we found that there is a shortage of interpretation on complicated link behaviors of contributors. While reflecting on this behavior, we realized that the existing organization of link (i.e. one source development unit directing to one target development unit) is not sufficient. To address this situation, we intended to reorganize links in accordance with the number of target development units. We introduced two structures as follows:

(i) *Single-target link*: A link whose source development unit directs to one target.
(ii) *Multi-target link*: A link whose source development unit directs to more than one target.

Meanwhile, we also corrected the definition of CTI in Sec. 4.3.1 to adapt new structures. *Duration* is the time interval from the create time $timestamps_c$ of the

earliest development unit d_e to the create time of the latest development unit d_l.

$$\text{duration}(l) = timestamps_c(d_l) - timestamps_c(d_e).$$

We present the results of experiments on single-target and multi-target links in Fig. 6. In Fig. 6(a), we observed that the number of multi-target links (average 58.63%) are relatively more than the number of single-target links (average 41.37%). It shows that when checking related development units, contributors should perform adequate review for all links. What is more, according to statistics, we assumed that the number of multi-target links is positively correlated with the age of project. The veracity of the speculation remains to be demonstrated by future research. The duration of multi-target links (344.63 days) is obviously longer than duration of single-target links (166.42 days). That is to say the more complex the link behavior is, the longer it lasts. For more efficient and effective development, contributors are expected to control link duration in a relatively short duration.

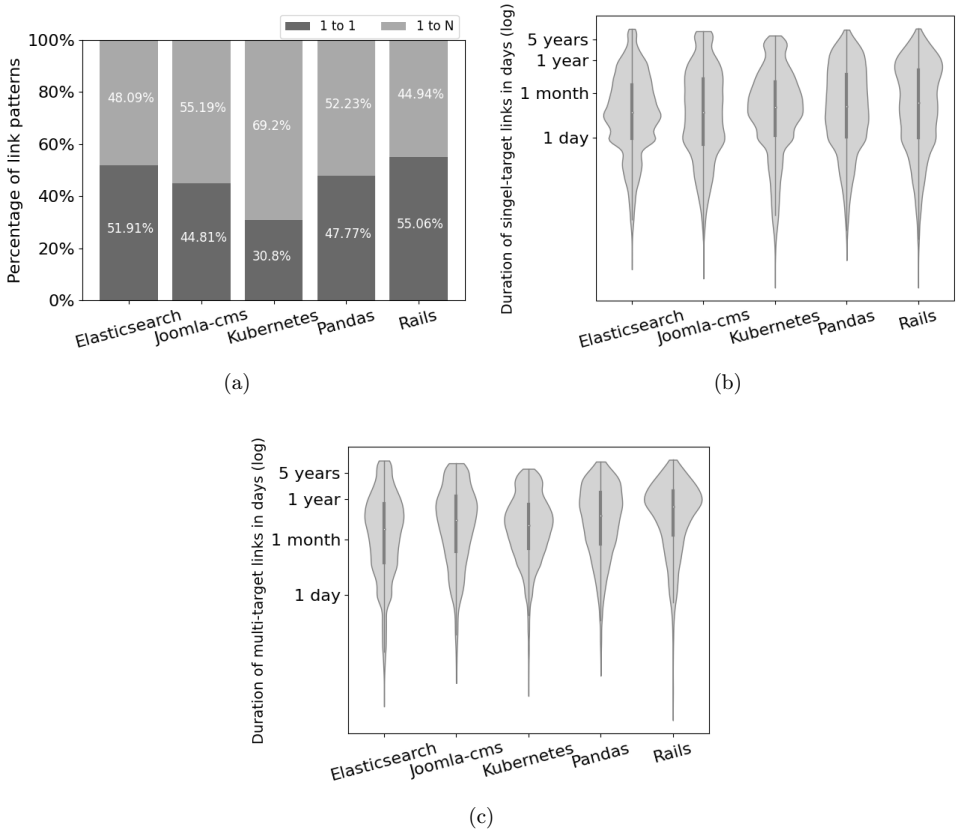


Fig. 6. Results of experiments on single-target and multi-target links: (a) percentage of single-target and multi-target links; (b) duration of single-target links and (c) duration of multi-target links.

4.4.2. *Cluster*

To gain more insight of how link works in practical development scenarios, we investigated crossly connected development units and defined them as a *cluster*. For instance, Fig. 7 shows a cluster that involves six links (i.e. six edges) and gathers four development units (i.e. development units numbered as 2224, 2219, 2080 and 2150) in *Rails*. We identified cluster of links that is expected to be the representation of resolution to more complex tasks involving several different development units.

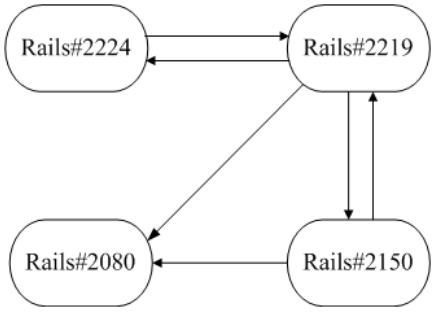


Fig. 7. Example of a cluster in *Rails* on GitHub.

We introduced the method we used in identifying cluster as follows. (i) Start a cluster from one development unit, and then find the development unit's target which is the first layer of cluster. (ii) Find out all target development units of each development unit in the first layer and they construct the second layer. (iii) Repeat step (ii) until all development units do not have any targets or the target is involved in former layers of the cluster already. We restricted that the number of cluster layers is at least 2 in order to distinguish from multi-target links. Overall, we got 11,227 clusters, as shown in Table 3.

Table 3. Statistics of the number of clusters.

Project	Elasticsearch	Joomla-cms	Kubernetes	Pandas	Rails	Total
#Clusters	3181	1384	4025	1913	724	11,227

For each cluster, we defined two metrics to reveal the complexity and scale of cluster: (i) *cluster depth* presents the number of layers of cluster and (ii) *cluster size* presents the number of development units in a cluster. Additionally, *duration* is also considered which is defined in Sec. 4.4.1.

Figure 8 illustrates the distributions of three metrics across studied projects. We observed that the distributions of cluster depth and cluster size are quite skewed towards the minimum (i.e. two layers and three development units). It shows that the size of most clusters is relatively small and it is possible to acquire awareness of

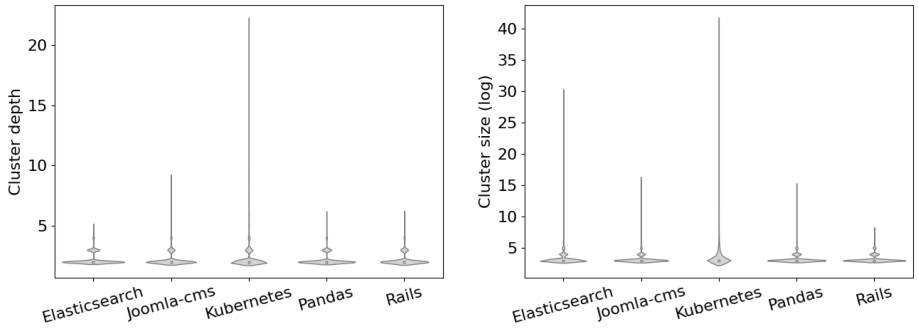
project by detecting clusters. However, *Kubernetes* is the project that has the most large-scale cluster (i.e. 22 layers and 41 development units). Notably, *Kubernetes* is also the project which has the least multi-target links in Sec. 4.4.1. It manifests that two structures (i.e. multi-target link and cluster) have reflected the usage of links on different sides. Furthermore, duration of clusters is on average 372.51 days which is a quite long time for contributors to review.

Summary. *In three projects, multi-target links are more common than single-target links, especially the ratio reaches 69.2% versus 30.8% in the project Kubernetes. The duration of a structure increases with the complexity of link structure, i.e. the duration of multi-target links is longer than that of single-target links, and the time span of clusters is longer than that of multi-target and single-target links.*

Implication. *We recommend that integrators should release fine-grained development tasks reasonably to decouple different development units, shorten the duration of issue resolution and reduce the size of cluster. It is reasonable for contributors to complete their contribution in a timely manner, and create links and specify the reasons when it is necessary to divide the contribution into multiple development units. It is a better choice for automatic tool designers to provide list-based results rather than threshold-based results. In addition, the visualization of link is also a potential functionality for tools to present complex structures of links.*

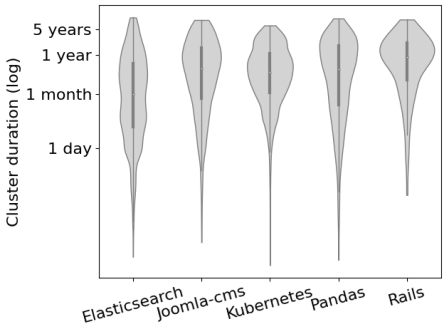
4.5. *RQ5: Why are links used?*

In this research question, we sought to understand why are development units linked. We conducted a card sorting on 500 randomly selected links. The sample yields a 90% confidence level with a 3.7% error margin. We focused on the context where developers built links and Table 4 shows the results distributed in four categories of link type.



(a) (b)

Fig. 8. Results of experiments on cluster: (a) statistics of cluster depth; (b) statistics of cluster size and (c) duration of cluster.



(c)

Fig. 8. (*Continued*)

Table 4. Overview of reasons in each type of links.

Reasons	P–P	P–I	I–P	I–I	Total
Bug fix	2	160	46		208
Collaboration	65			33	98
Supersede	51			12	63
Summary			5	30	35
Cherry pick	35			2	37
Backport	30				30
Duplicate	3			10	28
Defect report		2	18		20
Task assignment		3	4		7
Non-context	5	1			6
Block	2	2		1	5

The most common reason why practitioners use links is *bug fix*. Among reasons in P–I and I–P, bug fix accounts for 85.48%. It highlights the practicality of the pull-based development mechanism in promoting OSS development by solving issues. There are also 20 links aiming at *reporting defect* from existing PR. Around 19% of links are used for *collaboration* connecting different issues or PRs. These links aim to accomplish the same task and are divided into different parts. Correspondingly, out of 500 instances, seven links are used in *task assignment* by splitting a complicated task or contribution into small parts and accomplished by either one or more contributors. After the development stage, the practitioner, usually an integrator, creates a development unit to *summary* development status and maintain awareness of project. Contributors strive for effective and efficient development by figuring out a *duplicate* (5.6%) that contains two inadvertently identical development units. They also try to ensure successful integrating by identifying *block* links (1%), in which the resolution of one development unit is blocked by another development unit. A complementary reason is *backport* which accounts for 6%. It connects two contributions aiming at two versions of code. *Supersede* (12.6%) is also mentioned for development unit replacement. Finally,

1.2% of links are marked as *non-context* for lack of sufficient context to identify the reason. According to the reasons, we had several findings.

Efficient development. As suggested by multiple guidelines, before first attempting to create a contribution, contributor should announce the task she/he is working on and checks the project's current status. *Bug fix*, which acts on issue resolution directly, reveals the role pull-based development mechanism plays in promoting OSS development. While *defect report* raises questions about inadequate code review process resulting in defect in merged code.

Task and contribution division. Considering the continuous development nature of OSS, amount of tasks emerge all the time. *Collaboration* indicates a critical step before creating a contribution, which divides contributions into small patches. It is manifested by Gousios *et al.* [37] that the smaller size of code patch is easier to integrate. Correspondingly, practitioners also need to do *task assignment* and *summary* in time to maintain project awareness.

Resources waste. Even though many guidelines and researches suggest contributors examine existing contributions and the project's issue database, links of *duplicate* and *block* figure out redundancy of time and resources from both contributor's and integrator's perspectives. At the same time, links in *supersede* have both positive and negative impacts on development. The positive impact is the enhancement of code quality when competing with other contributors, while the negative impact is the waste of resources.

Continuous service. Approximately, 20% of links in P–P are used to identify *backport*. It underlines the importance of maintaining project service across different code versions. Contributors are suggested to take backport into consideration after the merge of a contribution.

Communication fatigue. As reported in [37], *non-context* reveals the paradox between the need of maintaining project's awareness and the behavior of rare communication from contributors. It hinders the observability of the overall status of a project and burdens integrators and contributors with extra communication obligations [37].

Summary. *In each link type, specific reasons of links and their distributions are very different, among which bug fix is the most popular reason. Moreover, reasons in P–P and I–I are more diverse and the most common reason is collaboration.*

Implication. *Contributors should specify the reason for creating the link in order to provide clues to knowledge organization in the complex context of links. It is also important for integrators to introduce formal usage of link, such as emphasizing the reason and categorizing the links according to the reasons. Automatic tool should also expand its functionality to identify the reason of links.*

5. Conclusion

In this paper, we conducted an empirical study on links among tasks and contributions in the context of pull-based development on GitHub. The goal of our study is to better understand the real usage and the influence of links in practice. We analyzed five popular GitHub projects. We found that most links are associated with PRs and used in obvious locations (e.g. documentation and comment). We also found that, on average, the lifecycle of a link is half a year and practitioners spend half a month discovering links. We observed that links have different structures (e.g. multi-target link and cluster) which are usually small-sized and possible to detect. We also observed that various reasons contribute to the usage of links in different contexts. Based on the findings of our study, we recommend that enough attention should be paid to link in OSS community for it contains rich knowledge about development and is helpful in maintaining awareness of projects to all practitioners publicly. Meanwhile, an automatic link recommendation tool is also suggested to be integrated into OSS platforms for intellective OSS development.

Our future work includes expanding the research to more projects, verifying the usefulness of initial findings, improving our findings to reduce researchers' biases, and looking for other factors that affect practitioners' linking behaviors. In addition, we will summarize the best practices of link usage, and evaluate the pros and cons of each mode in link usage through regression analysis. Combining the practical usage of links in different projects, we will explore formal methods of link usage and automatic link recommendation tools. Learning the process and characteristics of process and knowledge management in traditional software engineering may improve link usage in OSS projects.

Acknowledgments

This work was supported by National Grand R&D Plan (Grant No. 2020AAA0103504).

References

1. GitHub, Inc., The 2020 state of the Octoverse, https://octoverse.github.com/. Accessed 2021-9-2 (2020).
2. Y. Yu, G. Yin, H. Wang and T. Wang, Exploring the patterns of social behavior in GitHub, in *Proc. 1st Int. Workshop on Crowd-based Software Development Methods and Technologies*, 2014, pp. 31–36.
3. T. Wang, G. Yin, Y. U. Yue, Y. Zhang and H. Wang, Crowd-intelligence-based software development method and practices, *Sci. Sin. Inf.* **50**(3) (2020) 318–334.
4. H. Wang, Harnessing the crowd wisdom for software trustworthiness: Practices in China, *ACM SIGSOFT Softw. Eng. Notes* **43** (2018) 1–6.
5. S. Sowe, I. Stamelos and L. Angelis, Identifying knowledge brokers that yield software engineering knowledge in OSS projects, *Inf. Softw. Technol.* **48**(11) (2006) 1025–1033.

6. H. Wang, G. Yin, T. Wang and Y. Yu, *Crowd-Based Methodology of Software Development in the Internet Era* (Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability, 2019).
7. M. Golzadeh, A. Decan, D. Legay and T. Mens, A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments, *J. Syst. Softw.* **175** (2021) 110911.
8. G. Gousios, M. Pinzger and A. van Deursen, An exploratory study of the pull-based software development model, in *Proc. 36th Int. Conf. Software Engineering*, 2014, pp. 345–355.
9. M. I. Azeem, S. Panichella, A. Di Sorbo, A. Serebrenik and Q. Wang, Action-based recommendation in pull-request development, in *Proc. Int. Conf. Software and System Processes*, 2020, pp. 115–124.
10. M. Michlmayr, Quality improvement in volunteer free and open source software projects, Opensource, MIT (2007).
11. B. Lin, G. Robles and A. Serebrenik, Developer turnover in global, industrial open source projects: Insights from applying survival analysis, in *2017 IEEE 12th Int. Conf. Global Software Engineering*, 2017, pp. 66–75.
12. L. Dabbish, C. Stuart, J. Tsay and J. Herbsleb, Social coding in GitHub: Transparency and collaboration in an open software repository, in *Proc. ACM 2012 Conf. Computer Supported Cooperative Work*, 2012, pp. 1277–1286.
13. R. Kallis, A. Di Sorbo, G. Canfora and S. Panichella, Predicting issue types on GitHub, *Sci. Comput. Program.* **205** (2020) 102598.
14. C. Bird, A. Gourley and P. Devanbu, Detecting patch submission and acceptance in OSS projects, in *Fourth Int. Workshop on Mining Software Repositories*, 2007, pp. 26–26.
15. M. Aberdour, Achieving quality in open-source software, *IEEE Softw.* **24**(1) (2007) 58–64.
16. M. Rashid, P. M. Clarke and R. V. O'Connor, Exploring knowledge loss in open source software (OSS) projects, in *Int. Conf. Software Process Improvement and Capability Determination*, 2017, pp. 481–495.
17. Y. Zhang, Y. Yu, H. Wang, B. Vasilescu and V. Filkov, Within-ecosystem issue linking: A large-scale study of rails, in *Proc. 7th Int. Workshop on Software Mining*, 2018, pp. 12–19.
18. F. Zampetti, L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta and M. Lanza, How developers document pull requests with external references, in *2017 IEEE/ACM 25th Int. Conf. Program Comprehension*, 2017, pp. 23–33.
19. M. M. Rahman and C. K. Roy, An insight into the pull requests of GitHub, in *Proc. 11th Working Conf. Mining Software Repositories*, 2014, pp. 364–367.
20. Q. Fan, Y. Yu, G. Yin, T. Wang and H. Wang, Where is the road for issue reports classification based on text mining?, in *2017 ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement*, 2017, pp. 121–130.
21. L. Li, Z. Ren, X. Li, W. Zou and H. Jiang, How are issue units linked? Empirical study on the linking behavior in GitHub, in *2018 25th Asia-Pacific Software Engineering Conf.*, 2018, pp. 386–395.
22. M. Chen, D. Hu, T. Wang, J. Long, G. Yin, Y. Yu and Y. Zhang, Using document embedding techniques for similar bug reports recommendation, in *2018 IEEE 9th Int. Conf. Software Engineering and Service Science*, 2018, pp. 811–814.
23. Y. Zhang, Y. Wu, T. Wang and H. Wang, iLinker: A novel approach for issue knowledge acquisition in GitHub projects, *World Wide Web* **23** (2020) 1589–1619.
24. Z. Li, Y. Yu, M. Zhou, T. Wang, G. Yin, L. Lan and H. Wang, Redundancy, context, and preference: An empirical study of duplicate pull requests in OSS projects, *IEEE Trans. Softw. Eng.* (2020), https://ieeexplore.ieee.org/document/9174755.
25. J. Tsay, L. Dabbish and J. Herbsleb, Influence of social and technical factors for evaluating contribution in GitHub, in *Proc. 36th Int. Conf. Software Engineering*, 2014, pp. 356–366.

26. A. Lee, J. C. Carver and A. Bosu, Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: A survey, in *2017 IEEE/ACM 39th Int. Conf. Software Engineering*, 2017, pp. 187–197.
27. Y. Yu, H. Wang, G. Yin and T. Wang, Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?, *Inf. Softw. Technol.* **74** (2016) 204–218.
28. H. Rocha, M. T. Valente, H. Marques-Neto and G. C. Murphy, An empirical study on recommendations of similar bugs, in *2016 IEEE 23rd Int. Conf. Software Analysis, Evolution, and Reengineering*, Vol. 1, 2016, pp. 46–56.
29. R. Kipling, *Just So Stories for Little Children* (Oxford Paperbacks, 1998).
30. M. A. Jabar, R. Ahmadi, M. Y. Shafazand, A. A. A. Ghani, F. Sidi *et al.*, An automated method for requirement determination and structuring based on 5W1H elements, in *2013 IEEE 4th Control and System Graduate Research Colloquium*, 2013, pp. 32–37.
31. J. Zhu, M. Zhou and A. Mockus, Effectiveness of code contribution: From patch-based to pull-request-based tools, in *Proc. 2016 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 2016, pp. 871–882.
32. K. R. Lakhani and R. G. Wolf, Why hackers do what they do: Understanding motivation and effort in free/open source software projects (2003). Available at SSRN: https://ssrn.com/abstract=443040 or http://dx.doi.org/10.2139/ssrn.443040.
33. Y. Yu, G. Yin, T. Wang, C. Yang and H. Wang, Determinants of pull-based development in the context of continuous integration, *Sci. China Inf. Sci.* **59**(8) (2016) 1–14.
34. Z.-X. Li, Y. Yu, G. Yin, T. Wang and H.-M. Wang, What are they talking about? Analyzing code reviews in pull-based development model, *J. Comput. Sci. Technol.* **32**(6) (2017) 1060–1075.
35. K. Peterson, The GitHub open source development process (2013), http://kevinp. me/github-process-research/github-processresearch.pdf (visited on 05 November 2017).
36. J. Noll, S. Beecham and I. Richardson, Global software development and collaboration: Barriers and solutions, *ACM Inroads* **1**(3) (2011) 66–78.
37. G. Gousios, A. Zaidman, M.-A. Storey and A. Van Deursen, Work practices and challenges in pull-based development: The integrator's perspective, in *2015 IEEE/ACM 37th IEEE Int. Conf. Software Engineering*, Vol. 1, 2015, pp. 358–368.
38. Y. Yu, H. Wang, V. Filkov, P. Devanbu and B. Vasilescu, Wait for it: Determinants of pull request evaluation latency on GitHub, in *2015 IEEE/ACM 12th Working Conf. Mining Software Repositories*, 2015, pp. 367–371.
39. G. Gousios, M.-A. Storey and A. Bacchelli, Work practices and challenges in pull-based development: The contributor's perspective, in *2016 IEEE/ACM 38th Int. Conf. Software Engineering*, 2016, pp. 285–296.
40. Z. Li, Y. Yu, T. Wang, G. Yin and H. Wang, Are you still working on this an empirical study on pull request abandonment, *IEEE Trans. Softw. Eng.*, https://ieeexplore.ieee.org/document/9332267/.
41. D. M. Soares, M. L. de Lima Júnior, L. Murta and A. Plastino, Acceptance factors of pull requests in open-source projects, in *Proc. 30th Annual ACM Symp. Applied Computing*, 2015, pp. 1541–1546.
42. Y. Yu, Z. Li, G. Yin, T. Wang and H. Wang, A dataset of duplicate pull-requests in GitHub, in *Proc. 15th Int. Conf. Mining Software Repositories*, 2018, pp. 22–25.
43. L. Ren, S. Zhou, C. Kästner and A. Wąsowski, Identifying redundancies in fork-based development, in *2019 IEEE 26th Int. Conf. Software Analysis, Evolution and Reengineering*, 2019, pp. 230–241.

44. S. Balali, I. Steinmacher, U. Annamalai, A. Sarma and M. A. Gerosa, Newcomers' barriers... is that all? An analysis of mentors' and newcomers' barriers in OSS projects, *Comput. Support. Coop. Work (CSCW)* **27**(3) (2018) 679–714.
45. F. O. Bjørnson and T. Dingsøyr, Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used, *Inf. Softw. Technol.* **50**(11) (2008) 1055–1068.
46. I. Rus, M. Lindvall and S. Sinha, Knowledge management in software engineering, *IEEE Softw.* **19**(3) (2002) 26–38.
47. B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen and S. Li, Predicting semantically linkable knowledge in developer online forums via convolutional neural network, in *2016 31st IEEE/ACM Int. Conf. Automated Software Engineering*, 2016, pp. 51–62.
48. S. Yazdipour, GitHub data exposure and accessing blocked data using the GraphQL security design flaw, arXiv:2005.13448.
49. GitHub, GitHub Flavored Markdown Spec (2019), https://github.github.com/gfm/. Accessed 2021-9-2.
50. K. Blincoe, F. Harrison and D. Damian, Ecosystems in GitHub and a method for ecosystem identification using reference coupling, in *2015 IEEE/ACM 12th Working Conf. Mining Software Repositories*, 2015, pp. 202–211.