Within-Ecosystem Issue Linking: A Large-scale Study of Rails

Yang Zhang
National University of Defense Technology, China
yangzhang15@nudt.edu.cn

Yue Yu
National University of Defense Technology, China
yuyue@nudt.edu.cn

Huaimin Wang
National University of Defense Technology, China
hmwang@nudt.edu.cn

Bogdan Vasilescu
Carnegie Mellon University, USA
vasilescu@cmu.edu

Vladimir Filkov
DECAL Lab, University of California, Davis, USA
filkov@cs.ucdavis.edu

ABSTRACT

Social coding facilitates the sharing of ideas within and between projects in an open source ecosystem. Bug fixing and triaging, in particular, are aided by linking issues in one project to potentially related issues within it or in other projects in the ecosystem. Identifying and linking to related issues is in general challenging, and more so across projects. Previous studies, on a limited number of projects have shown that linking to issues within a project associates with faster issue resolution times than cross-project linking. In this paper, we present a mixed methods study of the relationship between the practice of issue linking and issue resolution in the Rails ecosystem of open source projects. Using a qualitative study of issue linking we identify a discrete set of linking outcomes together with their coarse-grained effects on issue resolution. Using quantitative study of patterns in developer linking within and across projects, from a large-scale dataset of issues in Rails and its satellite projects, we find that developers tend to link more cross-project or cross-ecosystem issues over time. Furthermore, using models of issue resolution latency, when controlled for various attributes, we do not find evidence that linking across projects will retard issue resolution, but we do find that it is associated with more discussion.

CCS CONCEPTS

• **Software and its engineering** → *Collaboration in software development*;

KEYWORDS

GitHub; Software ecosystem; Issue linking

ACM Reference Format:
Yang Zhang, Yue Yu, Huaimin Wang, Bogdan Vasilescu, and Vladimir Filkov. 2018. Within-Ecosystem Issue Linking: A Large-scale Study of Rails. In *Proceedings of the 7th International Workshop on Software Mining (SoftwareMining '18), September 3, 2018, Montpellier, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3242887.3242891

SoftwareMining '18, September 3, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5975-7/18/09…$15.00
https://doi.org/10.1145/3242887.3242891

1 INTRODUCTION

GitHub is a platform for sharing of code and ideas, characterized by unprecedented transparency in distributed software development, where developers can effortlessly follow, if not contribute themselves, to work across different projects. Such social coding has led to enhanced code reuse and collaborations; it is not uncommon to find prolific developers in GitHub contributing code across different projects, thereby bridging different communities [21]. As a result, code from an ever larger corpus is shared and reused [12], with many projects depending on the same libraries and components. This contributes to the co-evolution of projects and the formation of ecosystems [14], *e.g.*, Rails [9], where code across related projects becomes interdependent.

In addition to the benefits of reuse, life in an ecosystem of interdependent projects may also pose challenges, especially around issue resolution. Indeed, ecosystem-level knowledge about bug triaging [15] and fixing [5] must now be shared between developers within and across projects. And just like code in a project becomes entangled with code in the rest of the ecosystem to which it belongs, so do defects. One manifestation of this is that many open issues in one project get linked to related issues, in the same or different projects, as developers are tracing the root causes of a problem [18]. Prior work has found, in a handful of open-source projects part of the Python scientific computing ecosystem, that bugs linked across projects are considered more severe by developers, and take longer to resolve than those with within-project or no links [18], just as socio-technical theory of coordination [6] might predict (because of the increased coordination costs to keep track of cross-project activities). Similar results have been obtained among independent projects: identifying and linking to related issues is challenging [3], and the benefit of linking between dissimilar issues is often tenuous—linked issues take longer to resolve on average than unlinked ones.

However, for co-evolving, interdependent projects, *i.e.*, projects in the same ecosystem (exact definition in §3.1), one could also make the opposite argument. Indeed, a mature ecosystem can operate as a well-oiled machine; the code co-dependencies between projects facilitate information brokerage, thereby enhancing the developer community's transactive memory [19]. In turn, this may facilitate issue resolution.

Better understanding the ecosystem-level issue linking practices of developers, and how these might affect issue resolution, can immediately pay dividends in informing the development of ecosystem-level issue triage tooling. More generally, it can also shed light on socio-technical coordination challenges faced by developers *in an ecosystem.* To that end, this paper reports on a mixed-methods, quantitative and qualitative case study of Ruby on Rails (hereafter Rails), that aims to understand the relationship between linking practice and issue resolution *from an ecosystem perspective*. We chose Rails for our case study as it forms one of the largest and most active projects on GitHub, with a long history and an ecosystem comprising tens of thousands of interconnected projects [2]. Part of our study can be considered an extended replication of Ma *et al.*’s [18] study of cross-project issue linking in seven GitHub projects from the scientific Python community: we have similar research goals, but we use much larger data sets and statistical modeling techniques that account for covariates.

Specifically, we start by qualitatively studying linking outcomes to better understand the patterns in the linking practices and their potential effects on issue resolution. We find that:

- There are six notable categories of linking outcomes and they may have different effects on issue resolution.

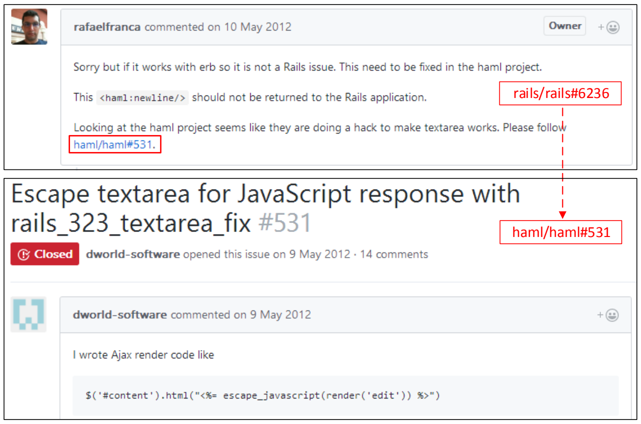
Guided by that, we then perform a quantitative study. We mine data from GitHub issues and build an approximation of the ecosystem of projects centered around Rails, linked to each other based on cross-referenced issues (via issue comments) within and across projects. Through analysis of link data, we investigate developers’ linking practices and their participation in issue resolution after the linking, as well as their evolution over time. Then, by separately regressing issue resolution latency and discussion length over variables of interest and various controls, we investigate how the linking practice affects the manners of issue management and resolution. The highlights of our quantitative findings are:

- Although developers tend to contribute most of their work within the project and ecosystem, interestingly, they show more diversity in linking practices by referencing more cross-project or cross-ecosystem issues over time.
- In contrast with the work by Ma *et al.* [18], we do not find evidence for increased resolution latency of issues linked to other issues outside the project.
- Issues linked outside of the project where they arise also have longer discussions.

Our data and scripts are online at https://github.com/yangzhangs/ecolinking_replication.

2 RESEARCH QUESTIONS

Issue resolution in GitHub is a collaborative and complex process. After a developer submits a new issue, other developers would discuss and evaluate this issue to see if it is worth fixing and, if so, to prioritize its fix. Linking can be an important step in the issue resolution process, as during their discussion stakeholders may link internally or externally to related issues, in order to provide what they consider are useful resources and information. The GitHub platform makes it easy to link to another issue or pull request from the same project or even different projects (see an example in Figure 1), via its Flavored Markdown technology. Eventually, helped by links and discussions, developers submit fix commits (*e.g.*, pull requests) for the issue, and it will be resolved and closed.



rafaelfranca commented on 10 May 2012

Sorry but if it works with erb so it is not a Rails issue. This need to be fixed in the haml project.

This `<haml:newline/>` should not be returned to the Rails application.

Looking at the haml project seems like they are doing a hack to make textarea works. Please follow haml/haml#531.

Escape textarea for JavaScript response with rails_323_textarea_fix #531

dworld-software opened this issue on 9 May 2012 · 14 comments

dworld-software commented on 9 May 2012

I wrote Ajax render code like

```
$('#content').html("<%= escape_javascript(render('edit')) %>")
```

Figure 1: An example of the issue linking in GitHub. In issue *rails/rails#6236*, developer *rafaelfranca* commented and linked to issue *haml/haml#531*.

To be helpful, a link to another issue should be timely and relevant to the current issue, as judged by a developer who posts it. But its placement in the discussion also depends on the practices and knowledge of the ecosystem’s interdependencies by the individual developers, which may vary between developers (*e.g.*, with experience), projects (*e.g.*, with project culture), and even as the issue progresses towards resolution (*e.g.*, with time), as more information about the root cause becomes available.

Based on the above discussion, it is reasonable to expect that different practices can lead to different outcomes in different projects. To gauge the different sources of variability in issue linking practices, and as ground work for the two research thrusts below, we start qualitatively by manually examining and coding a sample of issue pairs with links within and across projects in the Rails ecosystem:

Qualitative Study of Linking Outcomes: Identify the different linking outcomes during the issue resolution process.

Guided by the qualitative outcomes, we proceed along two research thrusts, the first aimed at understanding the linking practices of developers that lead to the different linking related outcomes, and the second aimed at understanding the factors affecting issue resolution latency and issue discussion length.

Research Thrust 1. Linking Practices. In this thrust, we consider spatial and temporal dimensions of issue linking:
RQ1-1. *Where do developers link to?*
RQ1-2. *How do these linking practices change over time?*

Research Thrust 2. Linking and Issue Resolution. Linking to a very related issue may enhance a discussion and even hasten the resolution of the original issue. But linking to a farther project in the ecosystem or an issue that has been long closed may provide diminishing returns. Having characterized the ecosystem level developers’ linking practice and involvement in terms of spatial and temporal factors, we proceed to investigate how different aspects of issue linking affect issue resolution time and discussion length. Specifically, we ask the following questions:
RQ2-1. *How do linking-related factors affect issue resolution?*
RQ2-2. *How do linking-related factors affect issue discussion?*

3 METHODS

3.1 Definition of the Rails Ecosystem

We adopt Lungu's [17] definition of software ecosystems, based on technical dependencies that exist between constituent projects. Blincoe *et al.* [2] proposed a new reference coupling method to detect technical dependencies, by using cross-references on GitHub, *i.e.*, cross-project links. Their study results showed that those links can be a good conceptualization of technical dependencies between projects, so our ecosystem definition is based on their study.

We define the *Rails ecosystem*, as of November 2016, as Rails together with all its immediately neighboring projects. Specifically, we call issues that have in their discussions links to other issues ***linked***, or ***source issues***, and issues that are linked to from other issues' discussions ***linked to***, or ***target issues***. We say that a pair of issues, S_A from project S and T_B from project T, are a ***linked pair***, or ***link***, (S_A, T_B), if the discussion on S_A contains a link to T_B. We also call projects S and T linked in that case, and call S the source project and T the target project.

Our operationalization, then, of the *Rails ecosystem* includes, besides Rails itself, two types of projects: (1) ***Type X projects***: all projects X such that the discussion of at least one issue A in Rails contains a link to an issue B in project X, *i.e.*, $Rails_A \rightarrow X_B$; and (2) ***Type Y projects***: all projects Y in which the discussion of at least one issue A contains a link to an issue B in Rails, *i.e.*, $Y_A \rightarrow Rails_B$, but no issue in Rails is linked to an issue in Y. Note that projects of type X may have issues that point back to issues in Rails, but no issues in projects of type Y are a target of Rails issues. Thus the sets of projects of type X and Y are disjoint. The time of our data collection was November 2016.

Under this definition, we can classify each link into three categories: (1) ***Within-project links***: source and target project are the same, and the project is part of the ecosystem. E.g., *rails/rails#1553→ rails/rails#1555*; (2) ***Cross-project links (Within-ecosystem links)***: source and target project are different, but they are both in the same ecosystem. E.g., *rails/rails#6236→haml/haml#531*, both of which we labeled as part of the Rails ecosystem; and (3) ***Cross-ecosystem links***: source and target project are different, and the target project is not in the ecosystem. E.g., *travis-ci/travis-ci#5562→nodejs/node-gyp#693*; while the former project does belong to the Rails ecosystem, as per our definition, the latter one does not. Note that both the within-ecosystem links and the cross-ecosystem links are links across projects.

3.2 Data Set

Identifying Rails Related Projects. We choose a large and popular project on GitHub, Ruby on Rails (Rails) and projects related to it as our case study. To identify related projects in the Rails ecosystem, first, we collected all Rails issues and their comments using the GitHub API before November 2016. By following the reference coupling method proposed by Blincoe *et al.* [2], connections between projects can be identified by matching the specific pattern "owner/repo#issueID" (*e.g.*, *rails/rails#1000*) in the comments of issues. We used regular expressions[1] to perform pattern matching and found projects with issues or pull requests mentioned in Rails (*i.e.*, Rails→Proj.X). Also, we used the "cross-referenced" GitHub API endpoint[2] to find all related projects that referred to at least one issue or pull request in Rails (*i.e.*, Proj.Y→Rails). This yielded 1,204 projects, including Rails, 281 projects with issues linked to from Rails issues (*i.e.*, type X above), and 922 projects with issues pointing to Rails issues (*i.e.*, type Y).

Table 1: Aggregate statistics of the 944 projects.

Statistic	Mean	St. Dev.	Min	Median	Max
#closed issues	317.7	893.3	1	72	11,401
#closed, linked issues	74.6	275.8	1	13	5,439
#links	124.6	525.1	1	17	11,347
#within-project links	103.2	449.5	0	12	10,032
#within-ecosystem links	9.3	41.6	0	2	908
#cross-ecosystem links	12.2	71.4	0	1	1,491

Collecting and filtering data. Through the GitHub API, we collected all data on issues, comments, and commits from all projects in our Rails ecosystem. We then collected data on links, including {*source_project, source_issue, actor, target_project, target_issue*, and *link_date*}, by using text parsing and the "cross-referenced" API. To focus on our analysis of the linking practices in GitHub issues, we only consider the source issues that are general issues not Pull Requests (PR). As for the target issues, we distinguish between PR and general issues (this information is kept in the *linkPR* indicator variable, see §3.4). Because we are interested in issue resolution latency, we only discuss issues that have been closed. Also, we only discuss those links in an issue that had occurred before the issue was finally closed. After this filtering, we obtained our final set of 944 projects, 253 with issues linked-to from Rails issues (type X), 690 with issues pointing to Rails issues (type Y), and Rails itself.

Basic descriptive statistics. In total, our dataset contains 284,087 closed issues. Among them, there are 70,395 (24.8%) issues that get linked and 114,185 linked pairs, or about 1.62 links per issue. 42.6% of target issues were pull requests and the other 57.4% were general issues. Table 1 presents aggregate descriptive statistics over the 944 projects in our dataset. We find that 82.8% of links were within-project links and 17.2% were across projects (within-ecosystem: 7.4%; cross-ecosystem: 9.8%).

3.3 Qualitative Analysis

To gain insight into the outcomes of the linking practices, we used open coding [10] to construct an inclusive set of linking outcomes. First, we randomly selected 120 linked issues from our data and removed one finally confirmed to be a duplicate, leaving 119 issues from 64 projects. Then, during a first round, one author carefully read the content of each sample and marked its keywords or statements. Next, we sampled issues and discussed them jointly by all authors. Later, we iteratively aggregated the descriptions and summarized the categories, led by one of the authors, again discussing and refining until reaching consensus.

3.4 Quantitative Analyses

Statistical Modeling. To discover relationships between linking practices and issue resolution, we developed two regression models, *Resolution latency model* and *Discussion length model*, by using

[1] github.com([a-zA-Z0-9-_.]+)/([a-zA-Z0-9-_.]+)/issues|pull/([0-9]+)

[2] https://developer.github.com/v3/issues/timeline/

multiple linear regression modeling (via function `lm` in R). In our models, we log-transformed variables where needed to stabilize their variance and reduce heteroscedasticity [8]. We removed the top 2% of the data to control outliers and improve model robustness. The variance inflation factors, which measure multicollinearity of the set of predictors in all our models, were safe, below 3. We use the adjusted R^2 statistic to evaluate the goodness-of-fit of our models. For each model variable, we report its coefficients, standard error, significance level, and sum of squares (via ANOVA analysis). Because each coefficient in the regression amounts to a hypothesis test, we employ *multiple hypothesis correction* over all coefficient results, to correct for false positives, using the Benjamini-Hochberg step-down procedure [1]. We consider the such corrected coefficients noteworthy if they were statistically significant at $p<0.05$.

Regression variables. The outcome (dependent) variables of the models are the issue resolution latency, ***issueLatency***, in days, and the discussion length in terms of the total number of comments, ***nComments***. There are several important stages during the resolution of an issue, as per the linking time of different link types. In our study, *Issue resolution latency* is the time interval between issue creation and its final (the last) closing date. And we only consider the resolution time of issues that are not PRs. Compared to the previous links, the last link may have a significant impact on the issue being closed, as, from an information theoretic perspective, all other things being equal and given the large size of the data set, the link before closing may have been most precipitative of the act of closing. Thus, we use the target issue of the last link to extract the linking-related factors.

In summary, our independent variables come from different confound areas: project-level, developer-level, and issue-level.

- ***teamSize***: number of contributors (who submitted at least one commit) in the project prior to the issue creation time. Larger teams may be better prepared to resolve issues;
- ***nIssues***: number of issues created in the project during the three months prior to issue creation, as context for the overall workload of the project;
- ***uContributor***: Binary, True if issue submitter has contributed code before the issue creation. This is a measure of previous interactions that issue submitter has had in the context of the project;
- ***nActors***: number of developers that participated in the source issue resolution, as a coarse-grained proxy for developer involvement;
- ***ratioM***, ***ratioCL***, ***ratioCM***, and ***ratioCML***: percentage of participants that do "M" ,"C+L", "C+M", and "C+M+L" activities in the source issue, as a fine-grained proxy for developer involvement[3];
- ***textLen***: total number of words in the source issue title and description text. Longer descriptions may indicate higher complexity of issue or better documentation;
- ***hasAssignee***, ***hasMilestone*** and ***hasLabel***: Binary variables to encode the presence of "assignee", "milestone", and "label" tags in the source issue, as a measure of the project team's responsiveness;
- ***isDifficult***: True if the source issue was predicted as "difficult", as a proxy for issue difficulty[4];
- ***linkLatency***: time interval between the source issue creation and the link creation, in days;
- ***linkClosed***: Binary, True if the target issue has already been closed when the last link was placed in the source issue;
- ***linkPR***: Binary, True if the target issue was a pull request;
- ***linkTextLen***: total number of words in the target issue's title and description text, as a measure of issue complexity;
- ***textSim***: text similarity (Cosine similarity) to the source issue, as a proxy for issue relevance;
- ***linkPlace***: types of links, we distinguished linking within the project, within the ecosystem, and across the ecosystem. We used effect coding [13] to set the contrasts of this three-way factor, *i.e.*, comparing each level to the grand mean of all three levels.

4 LINKING OUTCOMES

First, to better understand in what way linking to related issues helps resolve the source issue, we conducted a qualitative study of 119 pairs of linked, source-target issues. During our manual analysis, six categories of linking outcomes emerged.

[C1] **Closed internally:** without obvious help from the linked information (33.6% of samples). Developers linked to related issues either within the project or across projects, but those links did not obviously help close the source issue. E.g., in Sample 64 (*sass/sass#1212*), developer *tbremer* found what he thought was a related issue: *"I found this related issues that seems to be true for me: chriseppstein/compass#1488..."*. But developer *nex3* replied: *"This issue is about sourcemap generation, which seems unlikely to be related..."*. Finally, *nex3* closed the issue.

[C2] **Fixed internally:** after collecting useful information from linked issues (25.2% of samples). Developers found the target issue useful, *e.g.*, a fix already reported in the target repository, or some useful discussion, both of which can obviously hasten resolution time of the source issue. E.g., in Sample 79 (*twbs/bootstrap-sass#90*), developer *chrisnicola* provided two external links: *"There is a problem that has been happening with...compass/compass-rails#26, rails/rails#5497"*. But the issue's submitter *larryzhao* told him that the two links may not be helpful: *"I am not using compass for my project..."*. Then *chrisnicola* found another fix: *"The fix to sass/sass#337 actually fixes this completely for me..."*. *larryzhao* then promptly fixed his issue.

[C3] **Help fix target issue:** contribute to fixing link target issues, in other projects, after which the source issue can be readily fixed (10.9% of samples). Once developers find that the issue at hand is actually due to a problem in other projects, they try to reproduce it and issue a pull request to fix it, engaging in the type of altruistic/community behavior that OSS are famous for. E.g.,

[3] We manually checked 100 linked issue samples and categorized the involvement activities of developers into five categories:
- *Just Managed (M)*: developers did only management work on the issue, *e.g.*, adding a label, closing, and reopening.
- *Just Commented (C)*: developers just participated in the discussion by contributing comments.
- *Commented and Linked (C+L)*: developers commented, also they provided some links in their comments.
- *Commented and Managed (C+M)*: developers commented and performed a management action.
- *Commented, Managed, and Linked (C+M+L)*: developers commented, performed management action, and provided links.

[4] To quantify the ease of a source issue when it arrives, we first computed the mean resolution latency of ***unlinked issues*** and labeled those unlinked issues as "*difficult*" if their latencies are longer than the mean value, otherwise, as "*easy*". Next, we joined each issue's title and description text into a single text. By using Naive Bayes method, we trained our classification model on those issue text vectors. Then we gave each ***linked issue*** a predictive label ("difficult" or "easy"), representing the approximate difficulty that issue report with such text.

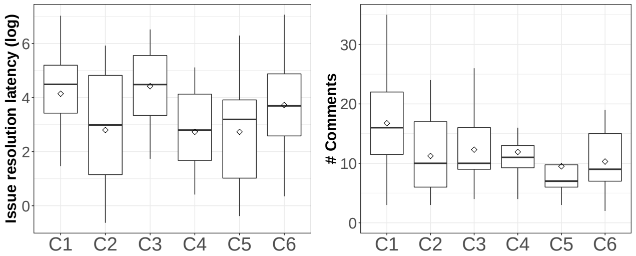


Figure 2: Resolution cost of different linking outcomes. *Left*: resolution latency (days); *Right*: discussion length.

in Sample 3[5], developer *timoschilling* commented: *"This couldn't be resolved by ActiveAdmin, it[s] a Ransack bug like activerecord-hackery/ransack#449..."* and he reported the issue to the Ransack project. After that, he made a pull request to fix this issue and the *activeadmin/activeadmin* issue also get resolved eventually.

[C4] **Closed with "not our problem":** wait for a fix in the target project and do nothing in the source project (10.1% of samples). When a developer found that an issue is actually due to a problem in other projects, they dropped it because they did not want to or could not fix it, or maybe they have time to wait for a fix in the other projects. E.g., in Sample 7 (*rails/rails#11843*), developer *thibaudgg* found an issue arising from the Squeel project: *"...are you using squeel gem too? I found this issue activerecord-hackery/squeel#265"*. Finally, *robin850* said: *"...this is issue seems related to Squeel so I'm giving it a close."* and closed this issue immediately.

[C5] **Closed with "won't fix":** duplicates; not urgent or important enough to fix (8.4% of samples). A developer would not fix an issue because they found this issue was a duplicate or has been already reported in other projects. Alternatively, the issue maybe was not urgent or important enough, so they closed it quickly. E.g., in Sample 73 (*travis-ci/travis-ci#6641*), developer *poplav* reported that he had the same issue in their repository, *"We have been running into these issues the past two days or so...Same as described above"*. And *ppires* pointed out that *"This issue is already open in sbt project: sbt/sbt#2758"*. *cotsog* closed the issue immediately.

[C6] **Closed but continued in a new issue:** done for technical reasons or perhaps these are complex issues that require more/renewed consideration (11.8% of samples). Developers found an issue is a difficult problem or they have spent a lot of time/resource on it, so they decided to open a new issue and moved further discussion to the new issue. E.g., in Sample 67 (*caskroom/homebrew-cask#3083*), after a long discussion, *vitorgalvao* provided a new issue and told other developers: *"We have a lot of text in this issue now, so it's becoming a bit unlikely that new eyes will go through all of it to see the discussion. There is a new issue where someone is offering to build this as an addition, so let's move the conversation there..."*.

Further, we compared issue resolution cost of each sample from this qualitative study. Figure 2 shows the comparison boxplots. We find that different linking outcomes may have different effects on issue resolution. E.g., issues from C1 need an average of 165.7 days (median: 83.8) to be closed, higher than the average latency of 70.7 days (median: 19.4) in C2 (Wilcoxon test; p=0.029). This is as expected, since having useful links may help. But it also indicates that there may be many false starts when searching related issues.

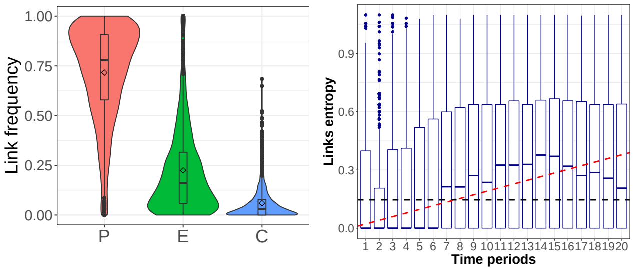


Figure 3: *Left*: Violin plots of the 3 different link types frequency distributions. *Right*: Links entropy, over time. Horizontal lines in boxplots: medians. Dotted horizontal line: overall median. Red dashed line: slope of the medians' slope.

> When developers link, there are different linking outcomes and they may have different effects on issue resolution.

However, the possible developer linking practices that can lead to the different linking outcomes have not been validated through extensive quantitative studies. We characterize them in our Research Trust 1 (see §5). Also, those linking practices may have different effects on issue resolution. We investigate them in our Research Trust 2 (see §6).

5 RT1: LINKING PRACTICES

5.1 RQ1-1: Linking contributions, summarized.

To find where developers link to more, we calculated the frequency of the 3 types of links (P: within-project links; E: within-ecosystem links; and C: cross-ecosystem links) that each developer contributed. We ranked all linking developers based on their total number of links contributed. Then we selected 1,703 developers who contributed more than 30 links each. We labeled each of their links as one of $\{P, E, C\}$. Then, for each developer, we separately calculated the percentage of each of the 3 types of links. Figure 3, left, shows violin plots of the link frequency distribution of the 3 types of links.

On average, 71.6% of links that each developer contributed were within-project links (median: 77.9%), 22.4% of links were within-ecosystem links (median: 16.1%), and 5.9% of links were cross-ecosystem links (median: 3.0%). We find that, *developers tend to link to more within-project resources, then within-ecosystem resources, and lastly cross-ecosystem resources.* We also analyzed the distribution of developers' code contribution, *i.e.* the proportion of their effort (via the number of commits) that is within the ecosystem. We find that on average, 64.7% of each developer's contribution is within the ecosystem (median: 66.7%). Thus,

> Developers tend to work within the ecosystem, consistent with their linking practices, *i.e.*, mostly linking to within-project and within-ecosystem resources.

5.2 RQ1-2: Evolution of linking practices.

It is reasonable to hypothesize that, over time, developers learn to link to related issues not just inside their project but also to other projects, both within an ecosystem and externally, so long as the linking is gainful. We next investigate this. Specifically, we ask if

[5] *activeadmin/activeadmin#3426*

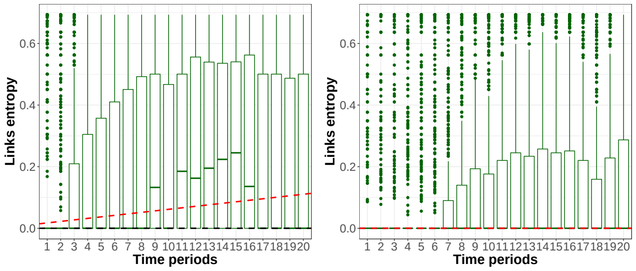


Figure 4: *Left*: The entropy evolution in within-project links and within-ecosystem links. *Right*: The entropy evolution in within-project links and cross-ecosystem links.

developers tend to increase the diversity of the types of projects they link to over time.

To assess the change in link type diversity, we selected the top-500 linking actors among all linking developers. Each of them contributed more than 150 links. For each developer d, we sorted all of his/her links based on their creation time. Each link has a type, one of $\{P, E, C\}$. We divided d's time from first to last link into 20 equal time periods, each corresponding to successive 5% intervals of their linking tenure. We used Shannon entropy to measure the link type diversity. If L_{ij} is the number of j-type ($j \in [1, 3]$) links that developer d contributed in the i-th ($i \in [1, 20]$) time period and L_i is the total number of links that d contributed in the i-th time period, then we define the *links entropy* of developer d in the i-th time period as: $-\sum_{j=1}^{3}\left(\frac{L_{ij}}{L_i}\right) \times log_2\left(\frac{L_{ij}}{L_i}\right)$.

Figure 3, right, shows the entropy changes over time of the different types of target projects (within-project, within-ecosystem, and cross-ecosystem). We observed that there is an upward trend in the entropy values over time, as the entropy values change from the 0 to 0.30, with some variability. This is consistent with a positive answer to our question, suggesting developers learn to link more broadly, linking outside of the project when no within-project related issues are apparent. Thus, *with time, the developers' linking tends to shift from single to multiple link types.*

Further, we separately looked at the induced sequences of just two types of links, (1) only within-project links and within-ecosystem links, and (2) only within-project links and cross-ecosystem links. Shown in Figure 4, we observed that in the former case, there is a slight upward trend in the entropy medians per period. In the latter case, the entropy values have large variance and no obvious trend (the median of each time period is 0). Thus,

Although developers tend to link to a broader set of project types over time, linking to resources in the same ecosystem dominates over linking to resources outside of the ecosystem.

6 RT2: LINKING AND ISSUE RESOLUTION

6.1 RQ2-1: Issue resolution latency.

We developed the *Resolution latency model* for investigating the relationship between issue resolution latency and link properties. Table 2 shows the modeling result. We can see that this model achieves a good fit (R^2=54.3%).

Table 2: Resolution latency model. The response is *log(issueLatency)*. R^2=0.54.

	Coeffs (Error)	Sum Sq.
(Intercept)	-0.1328 (0.0096)***	
log(nComments+0.5)	0.0079 (0.0038)*	6,719.6***
log(teamSize+0.5)	-0.0587 (0.0031)***	384.9***
log(nIssues+0.5)	-0.0663 (0.0031)***	56.7***
uContributorTURE	0.1316 (0.0060)***	399.7***
log(textLen)	0.0115 (0.0028)***	24.3***
hasAssigneeTRUE	0.0198 (0.0081)*	33.7***
hasLabelTRUE	0.2426 (0.0058)***	733.0***
hasMilestoneTRUE	-0.0183 (0.0072)*	10.7***
isDifficultTRUE	0.2742 (0.0176)***	111.0***
nActors	0.0509 (0.0035)***	280.1***
ratioM	-0.0904 (0.0033)***	2.2*
ratioCM	0.0225 (0.0031)***	729.0***
ratioCL	-0.0688 (0.0030)***	113.9***
ratioCML	-0.2460 (0.0036)***	2,094.9***
log(linkLatency+0.5)	0.6180 (0.0029)***	25,435.1***
linkClosedTURE	0.0362 (0.0092)***	5.2***
linkPRTRUE	-0.0826 (0.0061)***	162.4***
log(linkTextLen)	0.0334 (0.0030)***	77.8***
textSim	-0.0217 (0.0028)***	43.0***
linkPlace=within-project	-0.0265 (0.0050)***	14.8***
linkPlace=within-ecosystem	0.0324 (0.0076)***	14.8***
linkPlace=cross-ecosystem	-0.0059 (0.0068)	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

We find that, a one-unit increase in ***hasLabel***, results in the increase in issue resolution latency of 27.5%, holding all other variables constant. This is consistent with Cabot *et al.* [4], who found that on average it might cost more time to solve those labeled issues. ***nActors*** has a significant positive effect. This is consistent with the notion that *difficult issues may require more participants.* But it also indicates that *having many developers involved in issue resolution may be counterproductive.* All of the fine-grained involvement variables have significant effects on issue resolution latency, but differ among each other. Holding other variables constant, a one-unit increase in ***ratioCL*** corresponds to a decrease in issue resolution latency of 6.6%; a one-unit increase in ***ratioCML*** results in a decrease in issue resolution latency of 21.8%, holding all other variables constant. Thus, *a developer's increased participation in other involvement activities, especially **linking**, may decrease issue resolution latency.*

The linking-related factors are highly significant and cover nearly 68.8% of the variance explained. Specifically, the ***linkLatency*** is significant and has the most sizeable and positive effect on issue resolution latency. Thus, *faster linking, faster issue resolving.* ***linkPR*** has a significant, negative effect. This can be explained by that *linking to a pull request may bring fix codes.* ***textSim*** has a significant, negative effect. For one unit increase, the outcome decreases by 2.1%, holding other variables constant. However, we find that compared to the overall mean of all types of links (***linkPlace***), linking across projects (within-ecosystem or cross-ecosystem) has a negligible effect on issue resolution latency (only 0.04% of the variance explained). Thus,

We find no evidence that linking across projects has an effect on issue resolution.

6.2 RQ2-2: Issue discussion length.

Next, we analyzed the effect of linking on the length of the issue discussion, *i.e.*, the number of comments. Table 3 shows the *Discussion length model* summary. The model has a good fit to the data (R^2=52.9%).

Table 3: Discussion length model. The response is *log(nComments)*. R^2=0.53.

	Coeffs (Error)	Sum Sq.
(Intercept)	-0.0456 (0.0097)***	
log(teamSize+0.5)	0.0147 (0.0031)***	937***
log(nIssues+0.5)	-0.0033 (0.0031)	48***
uContributorTURE	-0.1302 (0.0060)***	1,114***
log(textLen)	0.1051 (0.0028)***	1,819***
hasAssigneeTRUE	0.1141 (0.0082)***	107***
hasLabelTRUE	0.1113 (0.0059)***	253***
hasMilestoneTRUE	0.1481 (0.0073)***	145***
isDifficultTRUE	-0.1210 (0.0178)***	22***
nActors	0.4976 (0.0030)***	17,681***
ratioM	-0.3420 (0.0031)***	4,825***
ratioCM	-0.0527 (0.0031)***	31***
ratioCL	0.0131 (0.0030)***	566***
ratioCML	-0.2069 (0.0036)***	1,584***
log(linkLatency+0.5)	0.0979 (0.0029)***	6,440***
linkClosedTURE	0.0633 (0.0093)***	148***
linkPRTRUE	0.0002 (0.0062)	1
log(linkTextLen)	0.0239 (0.0031)***	48***
textSim	-0.0662 (0.0028)***	629***
linkPlace=within-project	-0.0789 (0.0050)***	121***
linkPlace=within-ecosystem	0.0327 (0.0077)***	121***
linkPlace=cross-ecosystem	0.0462 (0.0069)***	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

We see that, a one-unit increase in ***hasLabel***, results in the increase in discussion length of 11.8%, holding all other variables constant. The developer involvement factors are significant and cover nearly 67.6% of the variance explained. All of the fine-grained involvement variables that relate to management activities have significant negative effects. Thus, *the higher the percentage of developers involved in management activities, the shorter the discussion.*

The linking-related factors are significant and cover nearly 20.2% of the variance explained. Specifically, ***linkLatency*** has a significant positive effect on issue discussion length. A 10% increase to link latency has the effect of increasing discussion length by 0.9%, holding all other variables constant. Linking to a closed issue has a significant positive effect on the discussion. For a one-unit increase in ***linkClosed***, the expected increase in the discussion length is 6.5%, holding other variables constant. Compared to the overall mean across all types of links (***linkPlace***), linking within the project has a significant negative effect on discussion length, while linking across projects (within-ecosystem or cross-ecosystem) has a significant positive effect. The coefficients show that linking across the ecosystem has a higher effect than linking within the ecosystem, holding other variables constant. This is consistent with the notion that *developers tend to discuss issues more if the needed resources are not within their projects.*

Linking across projects is associated with longer developer discussion.

7 DISCUSSION

7.1 Implications

For researchers. We found that, due to the interrelationships between the projects in the same ecosystem, developers tend to work within the ecosystem (§5.1), which may also affect their linking practices (§4, §5.2) and issue resolution (§6). Thus, our study motivates the need for considering ecosystem level research as soon as practicable, perhaps at the same time as a within project related issue research is being conducted. Our regression modeling results showed that factors related to linking, *e.g.*, target issue's type and state, are associated with issue resolution outcomes (§6). Whether this association is causal and how the issue resolution is affected by other linking-related factors should be further empirically evaluated, as improvements to issue resolution efficiency can have immediate practical impact.

For developers. Our study showed that an increase in the amount of linking activities is associated with a decrease in issue resolution latency (§6.1). Thus, developers may benefit from participating in more linking activities. We found that having many developers involved in issue resolution may be counterproductive (§6), perhaps due to the overhead in communication or due to the tragedy of the commons phenomenon. Smaller, more cohesive groups may be more effective for this task, although how that can be achieved is not clear. Efforts to enhance the bug tossing and triaging processes by making them ecosystem-aware may also help, and could be a direction for further study. We did not find evidence that linking across projects is associated with longer issue resolution latency (§6.1). Therefore, developers should not harbor the preconception that linking across projects will retard issue resolution.

For tool builders. Our study found that linking across projects is associated with more discussion compared to linking within the same project (§6.2). Tool support for effective linking within the ecosystem (ecosystem-level bug triaging tool) to get better ecosystem awareness may aid with longer discussions. We also found that labeling issues may be counterproductive (§6), as was found in some prior studies [4], although not in others [20]. Perhaps labels are often too short and too precise, and thus will not show up in broader searches, which most of the initial searches are like. Free form, unstructured descriptions may be more compatible with modern search technologies, and could be used in parallel with labels.

7.2 Threats to Validity

Clearly, our operationalization of the Rails ecosystem is just an approximation. It is based on a single recent snapshot of the issue report interdependencies in Rails and may change slightly if computed at a different time; moreover, it gives more weight to older projects, that likely have more issues and, therefore, more chances of being included. However, besides being computationally tractable, this operationalization does enable us to reasonably distinguish between projects that are *closer* and *farther* from the Rails core, which we expect may play different roles in the issue resolution process, *e.g.*, as developers may be differently aware of activity therein.

In our data, we have filtered out some copies of issues and ill formed links, however there still were some abnormal issues present that contain a few simple modifications but have a long resolution time. This may lead to some bias in our study.

8 RELATED WORK

8.1 Software Ecosystems

Source code forges, especially social coding platforms like GitHub, have been steadily changing Open Source Software (OSS) development. Over time, research emphasis of code repositories has shifted from single projects to complex software ecosystems in which projects are developed and co-evolve with each other. Still, developers increasingly participate in multiple projects [21] and move across projects in the ecosystem with relative ease [16]. This has also affected issue management: to realize and achieve individual project goals it is often necessary to properly manage the contextual interaction of tasks, activities, and interdependencies across multiple projects [7]. While the existing literature helps researchers and software practitioners gain a deeper understanding of software ecosystems, few studies have investigated cross-project issue linking practices, the only exception being the study by Ma *et al.* [18]. In that paper, the authors report on a small empirical study of cross-project correlated bugs in the scientific Python ecosystem. Their study was based on a manual inspection of 271 pairs of cross-project bugs and an online survey with 116 respondents. They focused on cross-project root causes for tracking and coordination between the upstream and downstream projects during bug fixes. Our work here is substantially different in two aspects. First, we consider a comprehensive set of issues in the Rails ecosystem, several orders of magnitude larger than theirs, enabling powerful quantitative modeling. Second, our goal is different, as we aim to understand the effects of common linking practices in the ecosystem on issue resolution.

8.2 Bug Fixing and Triaging

Bug fixing is a complex, iterative, and time-consuming process that involves the entire developer community and thus poses particular coordination problems [11]. The existing characterization of bug fixing is predominantly based on studies of large individual projects, *i.e.*, how to help developers better automatically resolve the within-project bug reports [15]. However, these characterizations may be insufficient in light of recent changes, because bug fixing can be propagated from one system to the other to share information about related bugs, *i.e.*, cross-system bug fixing [5]. Boisselle *et al.* [3] found that developers lose a median of 47 days of potential collaboration and users lose 38 days waiting for fixes already made in other distributions. Furthermore, in a recent paper, Zampetti *et al.* [22] investigated how developers document pull requests in GitHub with external references (Q&A forums, code examples, *etc.*). They found that even though external resources can be useful to learn something new or to solve specific problems, they are still rarely referred. These challenges prompted us to conduct a study of how cross-linking practices affect issue resolution latency in a large ecosystem.

9 CONCLUSION

Here we have studied ecosystem issue linking using qualitative and quantitative analyses of linking practices in the Rails ecosystem. Our qualitative study results indicate that there are varied linking outcomes and they may have different effects on issue resolution. Our quantitative study results find that developers show more diversity in linking practices by referencing more cross-project or cross-ecosystem issues over time. Interestingly, in contrast with prior work, we do not find evidence that linking across projects affects resolution latency, but we do find that it is associated with more discussion. Some possible reasons for that difference are: a) our study is based on much more data than the prior study, b) our methods included controls for various effects while the other study did not, and c) the prior study had done a manual selection of cross-linked bugs whereas ours was more automatic.

ACKNOWLEDGMENTS

The bulk of this work was produced while the first author was visiting DECAL at UC Davis. We thank members of DECAL for their comments and directions on this research. We also thank the anonymous reviewers for their valuable comments on earlier versions of this paper. This work was supported by National Natural Science Foundation of China (Grant No. 61502512 and 61432020), and China Scholarship Council.

REFERENCES

[1] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the royal statistical society. Series B (Methodological)* (1995), 289–300.

[2] Kelly Blincoe, Francis Harrison, and Daniela Damian. 2015. Ecosystems in GitHub and a method for ecosystem identification using reference coupling. In *MSR*. IEEE, 202–207.

[3] Vincent Boisselle and Bram Adams. 2015. The impact of cross-distribution bug duplicates, empirical study on Debian and Ubuntu. In *SCAM*. IEEE, 131–140.

[4] Jordi Cabot, Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Belén Rolandi. 2015. Exploring the use of labels to categorize issues in open-source software projects. In *SANER*. IEEE, 550–554.

[5] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. 2011. Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD. In *MSR*. ACM, 143–152.

[6] Marcelo Cataldo and James D Herbsleb. 2013. Coordination breakdowns and their impact on development productivity and software failures. *IEEE Transactions on Software Engineering* 39, 3 (2013), 343–360.

[7] Cecil Eng Huang Chua and Adrian Yong Kwang Yeow. 2010. Artifacts, actors, and interactions in the cross-project coordination practices of open-source communities. *Journal of the Association for Information Systems* 11, 12 (2010), 838.

[8] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. 2013. *Applied multiple regression/correlation analysis for the behavioral sciences.*

[9] Eleni Constantinou and Tom Mens. 2016. Social and technical evolution of software ecosystems: a case study of Rails. In *European Conference on Software Architecture Workshops*. ACM, 23–26.

[10] Juliet Corbin and Anselm Strauss. 1990. Grounded theory research: Procedures, canons and evaluative criteria. *Zeitschrift für Soziologie* 19, 6 (1990), 418–427.

[11] Kevin Crowston and Barbara Scozzi. 2008. Bug fixing practices within free/libre open source software development teams. (2008).

[12] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: cross-project code reuse in GitHub. In *MSR*. IEEE, 291–301.

[13] Alkharusi H. 2012. Categorical variables in regression analysis: A comparison of dummy and effect coding. *International Journal of Education* 4, 2 (2012), 202–210.

[14] James Herbsleb, Christian Kästner, and Christopher Bogart. 2016. Intelligently Transparent Software Ecosystems. *IEEE Software* 33, 1 (2016), 89–96.

[15] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving bug triage with bug tossing graphs. In *ESEC/FSE*. ACM, 111–120.

[16] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. 2011. The onion patch: migration in open source ecosystems. In *ESEC/FSE*. ACM, 70–80.

[17] Mircea Lungu, Romain Robbes, and Michele Lanza. 2010. Recovering Inter-project Dependencies in Software Ecosystems. In *ASE*. ACM, 309–312.

[18] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How do developers fix cross-project correlated bugs?: A case study on the GitHub scientific Python ecosystem. In *ICSE*. IEEE, 381–392.

[19] Christina Manteli, Bart van den Hooff, Hans van Vliet, and Wilco van Duinkerken. 2014. Overcoming challenges in global software development: The role of brokers. In *RCIS*. IEEE, 1–9.

[20] Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng. 2010. The impact of social media on software engineering practices and tools. In *FoSER*. ACM, 359–364.

[21] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. 2016. The sky is not the limit: Multitasking across GitHub projects. In *ICSE*. ACM, 994–1005.

[22] Fiorella Zampetti, Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, and Michele Lanza. 2017. How developers document pull requests with external references. In *ICPC*. IEEE, 23–33.