How do Developers Fix Cross-project Correlated Bugs?

A case study on the GitHub scientific Python ecosystem

Wanwangying Ma[*] Lin Chen[*‡] Xiangyu Zhang[†‡] Yuming Zhou[*‡] Baowen Xu[*‡]
[*]State Key Laboratory for Novel Software Technology [†]Department of Computer Science
[*]Nanjing University, China [†]Purdue University, USA
wwyma@smail.nju.edu.cn, {lchen, zhouyuming, bwxu}@nju.edu.cn, xyzhang@cs.purdue.edu

***Abstract*—GitHub, a popular social-software-development platform, has fostered a variety of software ecosystems where projects depend on one another and practitioners interact with each other. Projects within an ecosystem often have complex inter-dependencies that impose new challenges in bug reporting and fixing. In this paper, we conduct an empirical study on cross-project correlated bugs, i.e., causally related bugs reported to different projects, focusing on two aspects: 1) how developers track the root causes across projects; and 2) how the downstream developers coordinate to deal with upstream bugs. Through manual inspection of bug reports collected from the scientific Python ecosystem and an online survey with developers, this study reveals the common practices of developers and the various factors in fixing cross-project bugs. These findings provide implications for future software bug analysis in the scope of ecosystem, as well as shed light on the requirements of issue trackers for such bugs.**

Keywords- GitHub ecosystems; cross-project correlated bugs; root causes tracking; coordinate

I. Introduction

GitHub, currently the largest code repository in the open source world, provides a convenient way for software practitioners to collaborate and share code with one another [1]. Benefiting from its "social coding" capabilities, software development on GitHub has evolved from a single project to socio-technical ecosystems within which complementary applications and services are co-developed. They also co-evolve through the dependencies between projects and the interactions between practitioners. Since GitHub provides an issue tracking system for repositories to report and discuss bugs and other issues, the relations within a GitHub ecosystem are very likely to be reflected in the issues reported to various projects located in the ecosystem.

Consider the following example. A performance issue (with id #4259) was reported to *Astropy*. After inspecting this issue, an *Astropy* developer found that the cause of the bug might root in its upstream project *NumPy*. Then another developer who had contributed to both *NumPy* and *Astropy* pointed out that a *NumPy* issue (with id #6467) might be related. The two developers then took part in the discussion of the *NumPy* issue and sent a test case to *NumPy*. With the help of *Astropy* developers, the *NumPy* bug was fixed and finally closed after confirming that the test passed and the *Astropy* issue was resolved. Fig. 1 shows the fixing process of this pair of bugs. We call the two causally related issues reported to different projects **cross-project correlated bugs**.

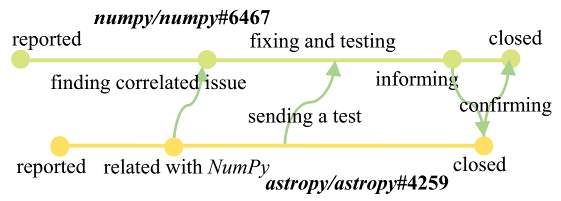


Figure 1. The fixing process of cross-project correlated bugs.

The above example is not uncommonly seen in practice. With the popularity of GitHub and the development of GitHub ecosystems, cross-project correlated bugs are increasingly important. New projects and practitioners joining GitHub often bring in new interactions within ecosystems, which may cause the ramifications of a bug to include more other projects, resulting in more cross-project correlated bugs. This phenomenon is confirmed by the responses of an online survey which we have conducted to investigate how developers view and deal with cross-project bugs (described in Section III.C). The results show that an average of 17.28% of the bugs that the respondents have encountered are cross-project ones. Compared with traditional (i.e., within-project and isolated) bugs, over half of the participants believe that cross-project bugs attract more concerns and cost more effort to deal with. In addition, though half of the respondents suggest that the severity of bug impact depends on multiple factors, 40.74% of the respondents still consider cross-project bugs generally more severe than within-project ones.

TABLE I. Statistical Comparision Between Cross-project Bugs and Within-project Bugs

		A	*I*	*M*	*N*	*P*	*SL*	*S*
Fixng time	*CB*	4.9	16.2	11.3	7.9	5.3	15.2	6.0
	WB	3.9	11.5	16.5	6.3	3.6	7.9	5.1
#Comments	*CB*	10.4	8.4	11.8	11.3	9.4	14.6	8.3
	WB	7.0	4.8	5.3	4.9	4.4	8.6	4.8
#Participants	*CB*	3.3	4.3	4.1	3.8	2.9	4.3	4.2
	WB	3.2	3.0	3.6	3.0	2.3	3.6	3.1

A**: Astropy, **I**: Ipython, **M**: Matplotlib, **N**: NumPy, **SL**: Scikit-learn, **S**: SciPy; **CB: cross-project bugs, ***WB***: within-project bugs; *Fixing time*: the duration between the reported time to the closed time of a bug (measured in 10^6 *seconds*), *#Comments*: the total number of comments in a bug report, *#Participants*: the total number of developers participating the discussion of a bug.
This table shows the mean value of each indicator. The numbers marked in grey indicate that the *p* value obtained from Wilcoxon rank-sum test is larger than 0.05, which means there is not a significant difference between cross-project and within-project bugs.

‡ Corresponding authors

Apart from the survey results, we also used the Wilcoxon rank-sum test to statistically compare the fixing time, number of comments, and number of participants between cross-project and within-project bugs in seven Python projects. The results listed in TABLE I show that the three values of cross-project bugs are significantly higher in most cases, implying that fixing cross-project bugs requires more effort. The survey responses and the statistical comparisons indicate that this emerging type of bug deserves more attention and investigation due to their significant quantity, severe impact, and difficulty in fixing.

The practices for repairing bugs have been studied for years. Aranda and Venolia identified the common bug fixing coordination patterns [2]; Guo et al. investigated factors that affected which bug could be fix [3] and characterized the bug reassignment process [4]; Breu et al. highlighted the importance of effectively and efficiently engaging user community in bug fixing activities [5]. These researches focused on the fixing process of within-project bugs. However, little work has investigated how developers fix cross-project bugs, especially what they do to deal with the following two challenges: 1) cross-project root cause tracking: as the bug carries over from one project to another, it becomes harder to trace the bug back to its root; and 2) fixing coordination: while waiting for an upstream fix, the downstream developers need to coordinate their project with the upstream one in order to minimize any undesirable impact of the upstream bug on their project.

The two challenges, which are not encountered in fixing within-project bugs, lead to high complexity in cross-project bug tracking and repair. Since there is a lack of empirical evidence in these aspects, this study aims to investigate how developers deal with cross-project correlated bugs, especially when facing the two challenges. We base our study on the scientific Python ecosystem, one of the most prominent ecosystems on GitHub [6]. Combining the manual inspection of 271 pairs of bugs and an online survey, we have two main findings. First, stack traces, communication with upstream developers, and familiarity with the involving projects are three main factors helpful for cross-project root cause tracking. Second, in spite of the extra maintenance burden, proposing a version-dependent workaround, i.e., a temporary and local solution is the most common practice of the downstream developers while waiting for an upstream fix.

The rest of the paper is organized as follows. Section II describes related work. Section III presents our research methodology, and Section IV shows our empirical results. We present further discussion on our findings in Section V, and examine threats to validity in Section VI. Finally, Section VII concludes this paper.

II. Related Work

A. Practices for Fixing Bugs

A number of literatures have investigated the developers' practices for fixing bugs by analyzing bug reports. Aranda and Venolia [2] used a combination of case studies and a survey of software professionals to investigate coordination activities during bug fixing. Guo et al. [3] described how the chances of fixing a bug was influenced by people's reputations, bug report edit activities, and geographical and organizational relationships. Later, Guo et al. [4] continued to analyze the bug reassignment process in Windows Vista and classified five primary reasons for reassignment. Breu et al. [5] found that active and ongoing participation of reporters was beneficial for a quick bug fix. Zimmermann et al. [7] characterized the bug reopen process using a mixed-method approach and quantitatively assessed the impact of various factors. These researches investigated how people communicated and coordinated to fix within-project bugs. Our study is different from them since we focus on the fixing process of cross-project bugs, which is significantly different from repairing within-project ones.

As for cross-project bugs, Canfora et al. [8] investigated Cross-System-Bug-Fixings (CSBFs) in FreeBSD and OpenBSD kernels and associated their occurrences to social characteristics of contributors using social network analysis. Their study paid attention to the identification of CSBFs from commits and the comparison of the roles of contributors involved and not-involved in CSBFs. However, our work aims at the developers' common practices of fixing cross-project bugs in GitHub ecosystems, which have not been analyzed in previous work.

B. Evolution of Software Ecosystems

Software ecosystems are a new and rapidly evolving phenomenon in the field of software engineering. Early analysis on software ecosystems mainly focus on defining the field and its limits [9–11]. In this study, we consider a software ecosystem as "a collection of software projects which are developed and evolve together in the same environment" [12].

Recently, a number of researchers have turned their eyes to the evolution of software ecosystems. Hora et al. [13] observed API evolution and its impact on Pharo ecosystem, as well as analyzed how developers reacted to API changes. Robbes et al. [14] investigated the ripple effects of API deprecations across a Smalltalk ecosystem. Bavota et al. [15] analyzed the evolution of project inter-dependencies in the Java subset of the Apache ecosystem. Decan et al. [16] studied the development of the R ecosystem on four repositories. They then further concentrated on inter-repository package dependency problems, discussing how GitHub influenced the R ecosystem [17].

Though these studies help researchers and software practitioners to gain a deep understanding of software ecosystems, few existing literatures have investigated the relationship between bugs in different projects, which also reflects the evolution of a software ecosystem. Our study makes a step towards this topic.

C. Collaboration in GitHub

GitHub, as the most popular web-based software repository, has attracted increasing attention from researchers during the last three years. Due to its prominent feature of social coding and pull-based model, a large number of research works on GitHub emphasize its social media [18, 19] and the collaboration between developers

[20–26]**.** Dabbish et al. [20] examined the collaborative utility of activity transparency based on a series of in-depth interviews with central and peripheral GitHub users. Tsay et al. [23] quantitatively analyzed how various technical and social measures were used to evaluate contributions in open source projects. Gousios et al. studied the practices and challenges in pull-based development from the perspectives of the integrators [25] and the contributors [26].

Our study also pays attention to the collaboration between GitHub developers. However, we concentrate on a specific aspect: cross-project bug fixing, which inherently involves the coordination between projects.

III. RESEARCH METHODOLOGY

In this section, we first describe the identification of cross-project correlated bugs in GitHub ecosystems. Then, we introduce the ecosystem and main projects under investigation. Finally, we present the research questions of our study and the corresponding research methods.

A. Identification of Cross-project Correlated Bugs

GitHub provides an issue tracking system for projects to record and follow the progress of any problem that their developers and users encounter during development, maintenance, or use [1]. An issue could be a bug, a requested feature, a task, a missing documentation, or even a question. In this study, we only focus on bugs that are faults in computer programs or systems causing the programs/systems to produce an incorrect or unexpected result, or to behave in unintended ways. Most GitHub projects use "labels" attached to an issue report to mark the different types of issues, such as "*defect*", "*Documentation*", and "*enhancement*" in *SciPy*. Thus, bugs in this paper only refer to the issues labeled with "*bug*" and its synonyms.

In this study, we define **cross-project correlated bugs** in GitHub ecosystems as follows: 1) they are a pair of issues reported to two inter-dependent projects located within the same ecosystem; 2) at least one of them is a bug; 3) an issue report refers to another report; and 4) one of the issues is the root cause of the other one.

In a GitHub issue report, there are two kinds of references between issues. First, if a participant mentioned another issue when commenting, he was very likely to leave an explicit link to it. For example, in astropy/astropy#4259, *mvhk* said "*I'd guess this slowdown is related to the recarray performance issues in numpy 1.10, i.e., numpy/numpy#6467*". Second, if an issue was mentioned in another issue's comment, a hint would show up automatically in this issue's page. Fig. 2 is a segment of numpy/numpy#6467's issue report page. This message indicates that esheldon/fitsio#58 referred to it. We parsed the issue report pages to identify the two kinds of references.

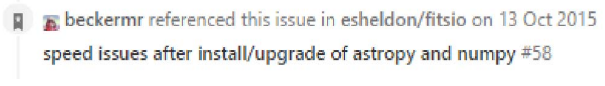

beckermr referenced this issue in esheldon/fitsio on 13 Oct 2015
speed issues after install/upgrade of astropy and numpy #58

Figure 2. Automatic references of other issues.

Specifically, we took the following four steps to identify cross-project correlated bugs. First, for any of the studied projects, we collected all its closed bugs. Then, for every bug *B*, we identified its related issues through both kinds of references to achieve a good coverage. Next, for *B*'s every related issue *R*, if it belonged to another project, *B* and *R* were regarded as a pair of candidate correlated bugs. Finally, we manually examined the issue reports of *B* and *R*. If *B* was caused by *R* or vice versa, they were considered as cross-project correlated bugs. For the simplicity of presentation, if *R* is the root-cause of *B*, then *R* is called the upstream bug and B is called the downstream bug, and vice versa.

B. Studied Ecosystem and Projects

In this paper, we investigated the cross-project correlated bugs in the scientific Python ecosystem. Python is one of the top three most popular programming languages used on GitHub in 2016 [27]. Augmented with a stack of open source tools for computation in mathematics, science, and engineering tasks developed by a diverse group of scientists and engineers, Python provides a computational ecosystem that is capable of tackling various challenges in scientific computing [28]. The work of Blincoe et al. [6] has shown that the scientific Python ecosystem is one of the most prominent ecosystems on GitHub with complicated dependencies among projects.

Particularly, we mainly explored seven GitHub hosted projects belonging to the scientific Python ecosystem. i.e, *IPython*[1], *NumPy*[2], *SciPy*[3], *Matplotlib*[4], *Pandas*[5], *Scikit-learn*[6], and *Astropy*[7].We selected these projects mainly for three reasons. First, they are the central packages in the ecosystem interacting with various scientific projects. Second, they are located at different layers in the scientific computing hierarchy, e.g., *IPython* is a working environment, *NumPy* and *SciPy* are tools providing primitives, *Pandas* provides high-performance data manipulation and analysis, *Matplotlib* is a supporting tool for data visualization, and *Scikit-learn*, and *Astropy* are designed for specific scientific fields (machine learning and astronomy, respectively [29]). Last, they are all active projects with a long life cycle, a number of experienced developers, and plenty of closed bugs.

TABLE II summarizes the basic information of the seven projects. To sum up, we have identified 542 pairs of candidate correlated bugs by parsing issue reports of the seven core projects. After manual inspection, 271 pairs were confirmed as cross-project correlated bugs (after removing 15 duplicate pairs). For the other 271 pairs of bugs with cross-references, the developers of one bug referred to the other bug because they could benefit from the discussion or the repair of the other one, or they misused the functionalities provided by the upstream project.

[1] http://ipython.org, https://github.com/ipython/ipython
[2] http://www.numpy.org, https://github.com/numpy/numpy
[3] http://www.scipy.org/scipylib, https://github.com/scipy/scipy
[4] http://matplotlib.org, https://github.com/matplotlib/matplotlib
[5] http://pandas.pydata.org, https://github.com/pydata/pandas
[6] http://scikit-learn.org, https://github.com/scikit-learn/scikit-learn
[7] http://www.astropy.org, https://github.com/astropy/astropy

TABLE II. Summary of Studied Projects Which Are Manually Reviewed

Project	owner	#devs	start time	#commits	#brs	#releases	#contributors	#bugs	bug label	#CBs
Astropy	astropy	22	20110730	14754	8	42	159	585	Bug	62
Ipython	ipython	13	20100510	21332	9	46	437	819	bug	36
Matplotlib	matplotlib	18	20110219	17453	5	49	434	209	confirmed bug	6
NumPy	numpy	13	20121020	14570	14	112	407	380	11 - Bug	49
Pandas	pydata	21	20100929	13506	3	70	520	2859	Bug	50
Scikit-learn	scikit-learn	37	20100831	20647	16	56	557	538	Bug	29
SciPy	scipy	22	20130426	15097	14	82	378	489	defect	54

The first two columns show the projects' names and their hosted organizations. The column *#devs* presents the number of members in the organizations. The column *start time* indicates when these projects started to use GitHub to track their issues. The data we collected for each project are from the corresponding start time (shown in the column *start time*) to March 12th, 2016. The next four columns show the total numbers of commits, branches, releases, and contributors at the time of March 12th, 2016. The columns *#bugs* and *#CBs* present the number of bugs and the number of identified cross-project correlated bugs in these projects during their study durations. The column *bug label* indicates the label that we used to select bug issues.

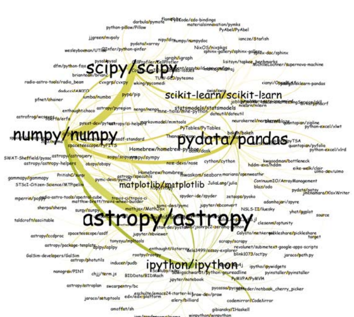


Figure 3. Part of the inter-dependencies among projects in the scientific Python ecosystem

In total, these correlated bugs involve a total number of 204 projects located in the scientific Python ecosystem. Fig. 3 shows the inter-dependencies between these projects. Each node stands for a project whose name follows the pattern *organization/repository*. The size of a node indicates the number of identified correlated bugs of this project. A directed edge from project P_1 to project P_2 means that a bug in P_1 referred to another bug in P_2. The thickness of an edge (P_1, P_2) means the number of pairs of correlated bugs which were reported to P_1 and P_2.

C. Research Questions

The aim of this study is to investigate the common practices of software practitioners during fixing cross-project correlated bugs. Compared with within-project bugs, the developers, especially the downstream maintainers, are facing the following two main challenges in the fixing process: 1) the cross-project root cause tracking; and 2) the coordination with upstream projects while waiting for a fix. Our study particularly focuses on the two aspects and attempts to answer the following three research questions:

RQ1: How long does it take to find the root cause of cross-project correlated bugs, that is, to link the downstream bug to the criminal upstream bug?

RQ2: What factors are important to tracking the root cause of cross-project correlated bugs?

RQ3: How do downstream developers coordinate with upstream projects to deal with cross-project correlated bugs after identifying the root cause?

D. Research Methods

To answer the three research questions, we first manually examined the issue reports of all the cross-project correlated bugs and summarized firsthand observations from an outsider's perspective. Then, we conducted a survey to explore how the insiders, i.e., the developers in scientific Python ecosystem, viewed and dealt with cross-project bugs.

1) Manual inspection of bug reports

The objectives of our manual inspection were two-folded: 1) to gain preliminary answers to the research questions from firsthand observations; and 2) to summarize the practices that we do not understand and want to inquire the developers when conducting the follow-up survey.

Three of the authors of this paper were assigned to this task. The manual inspection took two rounds and we inspected the bugs individually following the same procedure and criteria. During the first round, we 1) manually checked whether the candidate bugs automatically collected were cross-project bugs (by checking whether one of them was caused by the other); and 2) inspected the confirmed cross-project bugs as well as 350 random selected within-project bugs. The first round emphasized on finding out the main differences in the fixing processes between cross-project and within-project bugs, identifying the difficulties that the developers were facing, and summarizing the research focuses which we intended to make further investigation. During the second round, we looked into the 271 pairs of cross-project bugs to track the possible answers to the research questions and to explore promising information in the issue reports that could be quantitatively analyzed. At the end of either round, we got together to compare, discuss, and integrate our observations and ideas.

2) Online survey

Survey design. Since the upstream and downstream developers might have different concerns on the correlated bugs, we designed two questionnaires for the upstream (with nine survey questions, indicated by UQ with a question number in this paper) and the downstream (with ten survey questions, indicated by DQ) developers respectively. In particular, except for the last one which was an optional open-ended question, all the other questions of both questionnaires were multiple-choice questions. To further elicit the contributors' opinions, in most questions that had predefined answers, we included an optional comment box to encourage the respondents to provide the reasons of their choices or other options that we did not offer. The questions in the survey could be classified into three parts: 1) the developers' attitude towards cross-project bugs; 2) their treatment of fixing cross-project bugs; and 3) their problems and suggestions during the fixing process.

Target participants. We released the questionnaires on *SurveyMonkey*[8], one of the most famous web services for online surveys. The links of the surveys[9] were sent to the participants through emails. Our selected receivers consisted of two groups: 1) the developers who had ever taken part in the discussion of the collected correlated bugs (i.e., the reporters, commenters, and fixers); and 2) top contributors who had submitted at least one percent of the total commits for any of the 204 involving projects. We included the top contributors in our surveys because they care much about their projects and they are likely to pay attention to cross-project bugs even if they do not participate.

Respondents. In total, we have successfully sent invitation letters to 676 GitHub users and asked them to select a questionnaire suitable to them. Within a period of 15 days, we have received overall 116 responses (32 for the upstream questionnaire and 84 for the downstream questionnaire), with a response rate of 17.2%. It is acceptable since the response rate for online surveys in software engineering is usually within the 14–20% range [30]. Additionally, 43 participants replied the emails to show their interest in this research and to provide valuable suggestions and feedbacks for our work. Since we did not require the participants to fill in the comment field for any question, the numbers of received comments for different questions varied. Acting as supplementary data, totally 350 comments were received, with an average of 11 and 28 for each question in upstream and downstream questionnaires respectively.

Additionally, all the materials including the scripts, the collected data, the procedure of manual inspection, and the detailed description and analysis of the survey responses are available online[10].

IV. Research Results

This section presents the results of our research. When quoting survey respondents, we refer to the upstream and downstream participants using [URx] and [DRx] respectively, where *x* is the respondent's number.

A. RQ1:Difficulty of Finding the Root Cause

1) Observations from manual inspection

We propose the term *hidden duration* to measure how long the developers have spent in finding the root cause. It is defined as the duration from the reported time of the downstream bug to the linking time when one in a pair of cross-project correlated bugs was first referenced by the other one. Since the developers usually left an explicit link in the comment of the issue report once they identified the causal relationship between two bugs, the linking time could indicate when the root cause of the cross-project correlated bugs was found. The reported time and linking time were collected with the help of GitHub API[11] and parsing bug report pages. Fig. 4 shows the distribution of hidden duration for 271 pairs of cross-project bugs.

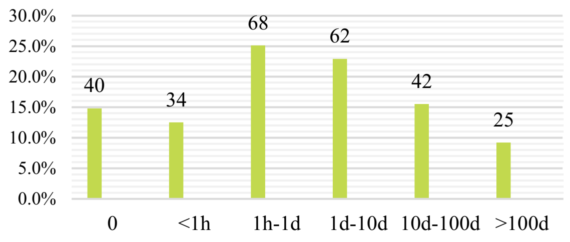


Figure 4. Distribution of hidden duration (h:hour, d:day)

Notably, the reporters of 40 (14.8%) downstream bugs knew the root causes at the time they submitted the issues (with a hidden duration of 0). However, from the bug reports, we are not able to know how long they had indeed spent in investigating the bug before reporting it. When reviewing the 40 bug reports, we find that 32 reporters are the downstream maintainers. They submitted a report locally rather than only discussed the bug in the upstream issue tracker mainly because they required a modification in their code to bypass the problem while the upstream side was working on a fix. For example, the reporter of astropy/astropy#4658 experienced a bug caused by *NumPy*. After reporting the problem to the upstream project, he also proposed, "*I think that should be eventually fixed upstream (numpy/numpy#7330) but I thought it might be worth it to include it here and later drop it again.*"

For the remaining 231 (85.2%) downstream bugs, the root causes of 102 of them were found in a relatively short time, i.e., less than one day. However, the remaining 129 bugs have longer hidden durations. Among them, 25 downstream bugs were linked with their corresponding upstream ones even after they had been reported for 100 days. After inspecting these bug reports, we find that the downstream developers might know where the problem was, but they were not sure about whether the upstream maintainers or they themselves should be responsible for fixing it. Therefore, they were hesitant to report their problem upstream. However, this is counter-productive as these bugs could have been be fixed in a much shorter duration if better coordination can be achieved between the upstream and downstream developers.

2) Survey responses

In order to better understand the perception of the downstream developers regarding cross-project bug tracking, we have made inquiries about whether it is difficult to find the root causes for cross-project bugs in the survey (DR2). Fig. 5 shows the survey responses.

[8] https://www.surveymonkey.com
[9] For upstream developers: https://www.surveymonkey.com/r/VSDMGSB
For downstream developers: https://www.surveymonkey.com/r/VWZX7L7
[10] https://github.com/njuap/ICSE2017
[11] https://developer.github.com/v3/

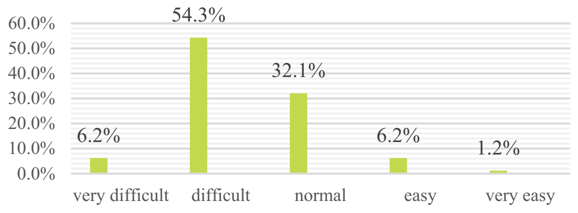


Figure 5. Responses to DQ2: is it difficult to find the root causes for cross-project bugs

We see that 54.3% and 6.2% of the respondents think of it as a difficult or very difficult task respectively, while 7.4% of them hold an opposite opinion since 6.2% choose "easy" and 1.2% choose "very easy". Another 32.1% do not have strong impression/opinions regarding the difficulty in cross-project tracking compared with within-project fault localization. Besides, several respondents confirm that the complexity of dealing with cross-project bugs lies in the root cause tracking. As DR7 comments, *"one has to make sure that the bug really comes from some other library and not from your own project."*

It takes more than one day to link nearly half of the studied downstream bugs to the corresponding upstream bugs; 60.5% of the surveyed downstream developers believe that tracking the root cause of cross-project bugs is difficult.

B. *RQ2:Important Factors for Tracking the Root Cause*

1) Observations from manual inspection

During inspection, we identify factors that may be useful for locating root causes of cross-project correlated bugs.

Stack traces. Previous studies have shown that bug reports containing stack traces, i.e., the sequences of calls that lead to the failure, will get fixed sooner [5, 31]. This seems also true for cross-project root cause tracking. After inspecting the 34 downstream bugs with a hidden duration less than one hour, we find that 24 of these bug reports contains stack traces which help the developers to narrow down the problem to an upstream bug.

Communication. For downstream developers lacking the knowledge of the upstream project, it is difficult to determine whether the unexpected behavior is just a feature of the upstream project or indeed a bug. Thus, communication with the upstream developers will be of great help. For example, when facing statsmodels/statsmodels#2668, a developer of *statsmodels* said, "*that's a bug that was introduced in numpy according the mailing list discussion (when I asked about this).*" Obviously, the contact with *Numpy* helped *statsmodels* find the root cause quickly. Thus, an automated tool that facilitates such communication will be highly desirable. For instance, communication/correlation with certain upstream projects/issues may be automatically suggested by correlating stack traces.

Familiarity. When fixing a within-project bug, the developer with expertise in the buggy component is the preferable assignee of the task, since he is familiar with the code and likely to fix it quickly [32, 33]. Similarly, while facing cross-project bugs, it is helpful to turn to the developers who specialize in the problematic component and know about the related upstream project. For example, when digging into a concurrency issue (scikit-learn/scikit-learn#4597), someone left a comment, "*ping @ogrisel @GaelVaroquaux, our parallelization experts?*" Then *ogrisel* who had also made significant contributions to the upstream project *joblib* quickly figured out the root cause of the issue (joblib/joblib#150). However, achieving this goal in the cross-project bug fixing context is more challenging than within-project bugs as cross-project expertise study needs to be performed.

2) Survey responses

When we ask the downstream developers what factors may act as positive roles to find the root-causes of cross-project bugs (DQ4), their responses confirm our observations. The stack traces, the communication with upstream developers, and the familiarity with the involving projects are selected by 73.8%, 72.6%, and 63.1% of the respondents respectively. Other factors that the developers specify include a simple and reliable test case provided by the reporter, the documentation of upstream project and comments in code, and discussion on *StackOverflow*.

Particularly, downstream developers emphasize the importance of communication. DR24 says in the comment box, "*One is rarely facile with the upstream project's internals, so communication is essential*". A number of the downstream respondents also suggest that prompt and clear communication between project developers is crucial for effective and efficient root cause tracking. Similarly, upstream developers also show positive attitudes towards the communication. The majority of upstream respondents note that they always (59.4%) or sometimes (37.5%) communicate with downstream developers (UQ3). The upstream practitioners believe that "*downstream projects are major users of upstream code*" [UR11] and downstream maintainers are in a position to understand when issues arise and if there are any implementation concerns when fixing them. As UR31 comments, "*I do this (communication with downstream developers) mostly for cross-project bugs since those people have more detailed knowledge about the concerned project and its code.*"

In summary, developers from both sides consider communication of great importance during fixing cross-project bugs. It is consistent with study on within-project bugs, as Ohira et al. pointed out that developers could fix bugs more efficiently through more discussions before assigning them [34]. However, when facing cross-project bugs, the upstream and downstream developers may focus on different aspects of the communication, leading to potential in-effectiveness. From the comments of the developers in our survey, we find that the upstream developers care more about the content. They mostly expect concrete description of the bug and clear description of the requirements of the downstream project, in order to understand the situation and start diagnosing the bug. On the contrary, the downstream developers are more concerned about the responsiveness of the upstream developers while underestimating the information needed by the upstream. They expect early and friendly responses, and several respondents note that the severity of the bug impact and the ease of fixing depend on "*how responsive upstream developers are to fixing the issue*" [DR73]. It is of importance to reach an agreement between the developers on both sides regarding the level of details in reporting downstream bugs and the timeliness of responses. Ideally, as long as the downstream developers report a relevant bug with the specified information, the upstream developers shall respond within a specific period. Admittedly, reaching such an agreement demands further study, as exposing too little from the downstream projects will be of no use whereas reporting too much may be counter-productive or even thwart the upstream developers.

The stack traces, the communication with upstream developers, and the familiarity with the involving projects are three main factors helpful for cross-project root cause tracking; Developers emphasize the importance of communication, but the upstream and downstream maintainers care about different aspects of the communication.

C. RQ3: Practices of Downtream Developers

1) Observations of manual inspection

During inspection, we find that the downstream developers may take different actions to deal with cross-project bugs after identifying the root causes.

Working round the bug locally. When the upstream developers are repairing the bug on their side, the downstream developers may propose a workaround, i.e., a temporary solution, to circumvent the problem locally and allow their project to continue working without waiting for an upstream fix. In 271 pairs of cross-project correlated bugs under investigation, 60 downstream bugs were closed with a workaround. Take scipy/scipy#4895 as an example. After a *SciPy* developer confirmed that it was caused by a *NumPy* bug (numpy/numpy#5895), he said, "*we need to work around it.*" Then *SciPy* got rid of that problem, but the underlying *NumPy* bug is still open.

Restricting the dependent upstream versions. Since an upstream bug is introduced in a specific version of the upstream project, usually the latest one, downgrading the upstream version turns out to be a simple way used by downstream projects to avoid the impact of the bug. For example, darbula/pymote#15 reported that a change in the newer version of *SciPy* (scipy/scipy#3125) broke the backwards compatibility, resulting in a failure of a *Pymote*'s functionality. Their developer suggested, "*Until scipy addresses this issue, one possible solution would be to fix scipy version to 0.12.*" During the inspection of bug reports, we find that eight downstream developers used this method to deal with cross-project bugs. This suggests the possibility of developing automatic technique to determine if a bug is cross-project and even to suppress the bug through systematically and automatic library downgrading.

Waiting for upstream fix. In some cases, the downstream developers may do nothing with their own project but wait for the fix of the upstream bug. This usually happens when the upstream maintainers are developing a new release and the downstream developers find a bug in the latest upstream development version. Since the bug is a pre-release one which does not appear in any officially released version, it will not have any severe impact on the downstream project as long as the upstream developers fix it before releasing the new version (e.g., the pair of cross-project correlated bugs: numpy/numpy#7393 and astropy/astropy#4677). For 49 pre-release cross-project bugs studied in this paper, the downstream developers chose to only wait for the upstream fix but made no modifications on downstream projects.

2) Survey responses

We also post a question in the survey to ask what the downstream developers usually do to deal with cross-project bugs (DQ7). The result indicates that proposing a temporary workaround is the most common practices, since 89.3% of the respondents choose it. Additionally, half of participants will restrict the dependent upstream versions to those without the corresponding upstream bug, while one third prefer to do nothing but wait for the upstream developers to fix it. A few developers (8.33%) choose to use an alternative upstream project providing the same functionality instead, which we do not notice during the manual inspection. For the four kinds of methods, DR24 indicates, "*whatever is easiest in their specific circumstances, above are good examples! but probably work around the issue*." Moreover, ten downstream developers in 16.7% of the respondents who select "other" specify that they will report the bug to the upstream project and/or help them to fix it if feasible. Next, we will discuss the use of workaround and the developers' attitude towards fixing upstream bugs with more details.

a) Workaround.

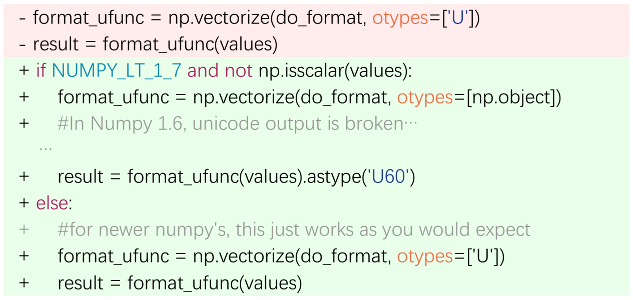
When fixing within-project bugs, developers may also prefer to choose a workaround rather than a real fix at the error-causing component. Murphy-Hill et al. summarized six factors that affect developers' decisions, that is, risk management, interface breakage, consistency, user behavior, cause understanding, and social factors [35]. In the context of cross-project bugs, workarounds are injected in the downstream projects to coordinate with the upstream projects. Combining the observations of manual inspection and survey responses, we find that two main reasons encourage the downstream programmers to develop workarounds for cross-project bugs.

Avoiding long-lasting impact. Usually, closing cross-project bugs in an upstream issue tracker does not mean the elimination of their impact on the downstream project. Until the upstream releases a version including the fix, the downstream project could be extricated. It is usually a long journey. As DR65 says, "*we need to wait for a release cycle to get the fix - sometimes a major release cycle.*" And DR27 also complaints, *"waiting for an upstream fix delays the release of a downstream fix.*" Then, "*a workaround must be implemented if the upstream team is not willing or able to fix the bug quickly*" [DR52], and it allows the downstream project to temporarily suppress the upstream bug.

Shielding the end users. As many users may still use an old version of the upstream project, the downstream developers cannot rely on a fix in the next upstream release.

As DR37 indicates, "*within project we control releases, but with dependencies a user might have an old version installed, so we have to work around bugs regardless of if they're already fixed upstream.*" So adding a workaround for an upstream bug enables the downstream project to support buggy upstream version without affecting the end users. An example is shown in scipy/scipy#3596.

However, though the downstream developers are willing to propose workarounds, they also complain about the extra maintenance burden of workarounds. In our survey, the downstream respondents note that they prefer to inject a workaround only in the situation when the bug exists in the upstream project, which means adding version-dependent code to the project. Fig. 6 shows a workaround in *Astropy* for a *NumPy* bug. Code was modified to support the buggy upstream versions (< *Numpy* 1.7), but the upstream method was used as expected for non-buggy versions.



```
- format_ufunc = np.vectorize(do_format, otypes=['U'])
- result = format_ufunc(values)
+ if NUMPY_LT_1_7 and not np.isscalar(values):
+     format_ufunc = np.vectorize(do_format, otypes=[np.object])
+     #In Numpy 1.6, unicode output is broken…
  …
+     result = format_ufunc(values).astype('U60')
+ else:
+     #for newer numpy's, this just works as you would expect
+     format_ufunc = np.vectorize(do_format, otypes=['U'])
+     result = format_ufunc(values)
```

Figure 6. A version-dependent workaround in the file *astropy/coordinates/angles.py*

In a word, when the downstream developers intend to work around cross-project bugs locally, "*you have to coordinate workarounds in your own code with releases of another project*" [DR83]. To achieve it, the downstream developers should keep up with the latest status of the upstream bugs, in order to get informed of which upstream release incorporates the fix and decide when to undo the workaround. It is absolutely an annoying process and "*for within-project bugs this is not necessary since all parts can be updated simultaneously*" [DR53]. Therefore, many respondents in our survey believe that the version-dependent workaround makes cross-project bug more difficult to cope with. However, the difficulty also suggests that automated tools to support (automatic) synthesis and (automatic) maintenance of workarounds would be highly desirable. For example, annotation can be inserted with a workaround and a notification channel can be established between the upstream and downstream projects, so that when the corresponding bug is fixed by the upstream, the annotation and automatic notification can help removing the workaround.

b) Reporting and fixing in the upstream project

We analyze the identities of the reporters and restorers of the upstream bugs in the downstream projects and show the results in Fig. 7.

The figure indicates that 97 (35.8%) of the 271 upstream bugs were reported by the owners of the downstream projects (i.e., the members of the organizations in charge of the downstream projects). In addition, 43 (20.5%) of the 210 upstream restorers who were confirmed by our manual inspection are also the downstream owners. The results confirm that after identifying the root causes, the downstream developers may report and help to fix the cross project bugs on the upstream side.

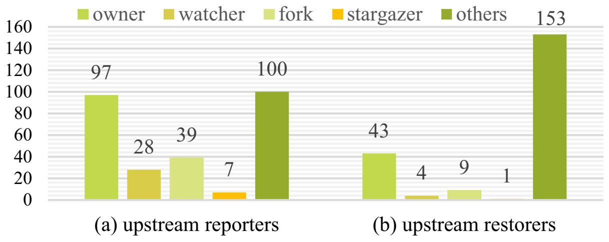


Figure 7. The identities of reporters and restorers of the upstream bugs (owner: owning the downstream project; watcher: receiving the notification of each event happening in the downstream project; fork: copying the downstream project to his account; stargazer: subscribing to the downstream project without receiving notifications)

In our survey, we also post a separate question to downstream developers about their altitude towards helping the upstream side to fix bug (DQ9). Nearly all of the respondents are always (35.8%) or sometimes (61.7%) willing to help the upstream developers to repair cross-project bugs. We classify the reasons into four categories.

- ***Dedication***. In the open source world, fixing an upstream bug helps everyone in the upstream and all its affected downstream projects, just as DR6 says, "*Open source, collective benefit.*"
- ***Obligation***. In some developers' opinion, in some sense, fixing cross-project bugs is also the responsibility of downstream projects, since they have chosen to adopt the upstream. DR59 points out, "*They (the upstream developers) don't really owe me anything. If I'm affected, I should be willing to put in effort too.*"
- ***Self-interest***. The ultimate solutions in the upstream project will relieve the downstream projects from the pain of cross-project bugs. Therefore, helping the upstream developers, as DR34 indicates, "*makes my project work better.*"
- ***Sense of honor and personal interest***. Other downstream developers contribute to an upstream fix because they think that, "*It's fun to battle the hard bugs and get to know new code bases*" [DR41], and "*I also like when my Name is in contributors list of some serious project*" [DR39].

However, though almost every downstream developer is ready to volunteer for upstream bug repairing, sometimes they are not able to help. The respondents point out that the main challenge that hinders the downstream developers to make contribution is their lack of the expertise of upstream projects. DR37 says, "*It's difficult to jump into a project (even one you use a lot) and learn their code, PR process, testing systems, etc. So there's a big learning curve, but it feels great to contribute to a big project you use a lot.*" Therefore, to encourage downstream collaboration and contribution, the upstream developers are supposed to keep their projects easy to learn and their maintenance practices easy to follow. A simple and direct way is to maintain concise documentation with outstanding focal points, including user manual, developer guide, and reference guide. Additionally, clear and well-written code comments are also important and helpful.

Downstream developers take different actions to deal with cross-project bugs after identifying the root causes; In spite of the extra maintenance burden, proposing a version-dependent workaround is the most common practice.

V. Discussions

A. Dilemmas in collaboration and coordination

From the empirical results, we can see developers emphasize collaboration and coordination during cross-project bug fixing. However, the different requirements of the upstream and downstream projects may place the developers in dilemmas, which post new challenges to tool design. Next, we will discuss the dilemmas that we find during analyzing bugs reports and survey responses.

The first dilemma is about cross-project testing. To prevent cross-project bugs, downstream developers in our survey note that it would be helpful if the testing suites for downstream projects are run before releasing an upstream version. However, the upstream developers have conservative opinions about cross-project testing due to the following difficulties. First, it is often impossible to get the complete list of downstream projects, and developers even may not know how their projects are used by the downstream ones [14]. Second, each project has a different way to run its tests. Last, it is extremely time consuming and the efforts may not pay off. Additionally, both the upstream and downstream sides complain about a lack of tools supporting effective cross-project testing. Hence, there is a demand to develop tools for cross-project testing to help the developers gain a tradeoff between preventing more cross-project bugs and excessive overhead.

The second dilemma concerns the notification of bug fixes. In order to deprecate outdated workarounds, the downstream developers have to know when the upstream bugs get fixed. Though GitHub users could subscribe to their interested issues to track the issue status and receive notifications of every new comment, they may easily be overloaded with a large volume of messages/notifications [36] and miss the information of bug fixing. Therefore, the downstream developers generally believe a notification of the bug fix from the upstream side would be nice. However, the notification should not be an extra burden for the upstream developers. A better way is to improve the notification scheme of GitHub so that it can send automatic massages to the participants of the linking issues in a separate thread when the upstream bug is set "closed" and the bug fix is released.

The last dilemma we discuss is the requirements for releasing the bug fix version. The downstream developers are always waiting for a new upstream release with the corresponding bug fixes, and they tend to upgrade their dependencies when the upstream side perform bug fix activities [15]. However, "*release cycles of downstream and upstream projects are out of sync*" [DR65], which in our survey is one of the most mentioned reasons why it is more difficult to deal with cross-project bugs in order to eliminate the undesirable effects on the downstream projects. Obviously, the downstream projects care much about when the upstream developers deliver a new version and hope for a quick release. However, in most cases, the upstream developers will "*make releases when they are ready*" [UR30]. They are cautious about every change to their code in the new version, and they need to take everything into consideration before they decide to include a bug fix in an upcoming version. Therefore, there is a conflict between the upstream and downstream projects: the former prefers to give a bit of time for reflection, while the latter requires an immediate release. This also poses new challenges in release management, especially for upstream projects. The release managers have to achieve a balance in the needs between themselves and their downstream projects. Above all, what UR32 suggests may be the ultimate goal: "*The reformation should help best of both-ends.*"

B. Implications for Tool Support

In the context of ecosystems, projects are not isolated and the developers are supposed to consider their neighbors while making decisions if they want their project to become more popular. Thus, ecosystem-wide tool support is needed to help the developers to deal with cross-project bugs.

1) Dependency registration

From the survey responses, we find that the upstream developers are facing an annoying problem: they are unaware of which projects are depending on them, so they cannot predict which projects will be influenced when a bug appears.

A service of dependency registration incorporated into GitHub may be helpful. By adding a "*depend*" button on the project homepage like "watch", "fork", and "star", the downstream projects can easily inform the upstream of their dependences. Then the upstream project can obtain the list of projects that uses it in its dashboard and stays aware of the relation with others. It would be a start to improve communication between projects.

Furthermore, through the known inter-relations between projects, the network of their located ecosystem can be easily constructed. Then, the visualization of the whole ecosystem will help developers to learn what position their projects are located in and evaluate how important and popular their projects are among others.

2) Bug referencing system

The ability of GitHub issue tracker to cross-link bugs in different project is considered a great help to deal with cross-project bugs. A considerable number of respondents suggest that if GitHub could provide a bug referencing system, it would be very useful to address cross-project bugs. Combining our observations and the developers' requirements, we propose that the system should have the following functions.

- Bug dependency visualization. Instead of an inline hint "*This was referenced……*" shown in Fig. 2 to indicate the interconnections between bugs, a reference graph is more direct and visible. More importantly, the graph could provide a global view of the whole issue dependency network with indirect and multi-level relationship. It can help the upstream developers to estimate the severity of a bug from an ecosystem-wide perspective. In addition, the downstream developers can easily find sibling projects suffering from the same upstream bug, in order to communicate and learn the workarounds from each other.
- Centralized tracking. The system is supposed to support a centralized tracking of all related bugs in a single web page. When browsing a bug report, it allows users to simultaneously read other reports without jumping to a new page by simply clicking the identifiers in the bug referencing graph. The information of different bug reports is expected to be restructured in a way that is convenient for the developers to compare ideas and trace the fixing process. The centralized tracking helps to include all relevant people in the discussion easily and allows direct communication between the developers from multiple projects without extra burden.
- Voting. Instead of leaving a message like "*+1*" in the comment, the system is supposed to include a voting component to show the support and opposition directly. Many users of GitHub issue tracker have suggested this feature [37, 38]. It is a more explicit way to collect opinions about comments and allows the readers to sort the comments by votes, which helps the maintainers to decide their way to fix cross-project bugs.
- Automatic notification. Important activities, such as fixing and closing a bug, should be automatically notified to the anticipating developers. Notably, the notified events should be carefully selected by the system or customized by the developers to prevent the developers from being lost in overloaded information.

As discussed earlier, our study also suggests the potential tool support on automated workaround synthesis and maintenance, and on automatic cross-project bug identification and suppression by systematic library downgrading.

VI. Threats to Validity

In this section, we discuss the threats to validity of our study. First, we may miss cross-project correlated bugs when collecting data. We identified the correlated bugs by picking up the explicit links in the comments and the automatic hints in issue pages. However, some commentators might just leave an implicit link, by saying "see here" with a hidden link associated with the word "here", leading to the omission of cross-project bugs. However, we also collected data using the automatic hints and inspected the issue reports manually, which maximized the accuracy of the investigated dataset. Additionally, some projects are just mirrored on GitHub, or prefer an independent tool to the issue tracker included, which might also cause underreporting in cross-project bugs.

The second threat concerns the researchers' preconceptions. The three authors that conducted the manual analysis followed the same procedure and criteria in determining cross-project bugs, time required to fix bugs, related interactions between developers, bug resolutions, and also in analyzing the survey results. However, it is in general difficult to completely eliminate the influence of researchers' preconceptions. In order to minimize personal bias, we crosschecked each other's results and discussed unclear cases together.

The third threat is that the skewness of the survey responses is inevitable. Since the respondents were voluntary to take part in the survey, the developers with spare time or who cared about cross-project bug fixing were more likely to participate. Consequently, results might be skewed towards those people. To encourage responses, the amount and type of the questions in the questionnaires were carefully designed according to [30].

The last threat concerns the generalization of our empirical results. We conducted our study on the scientific Python ecosystem. However, cross-project bugs do not only occur within the specific ecosystem. Decan et al. reported that the developers in R ecosystems felt it more and more of a pain if the upstream packages broke [17]. Holmes and Walker also described similar problems in other platforms [39]. We cannot assume that our results generalize beyond the specific environment where they were conducted. Further validation on other ecosystems in and outside of GitHub and even non-open-source ecosystems is desirable.

VII. Conclusion and Future Work

In this study, we investigated how developers fix cross-project correlated bugs, i.e., a pair of causally related issues in different projects within a GitHub ecosystem. In particular, we focused on two aspects: cross-project root cause tracking and coordination between the upstream and downstream projects during bug fixing. We manually identified and inspected 271 pairs of cross-project bugs in the scientific Python ecosystem and conducted an online survey with 116 respondents in the scientific Python community. Our empirical results reveal the common practices of developers when fixing cross-project bugs and provide suggestions for ecosystem-wide tool supports. In the future work, we intend to put efforts in two directions: effective and efficient multi-project testing for cross-project bugs and bug fix release management in software ecosystems.

Acknowledgment

We are grateful for the respondents who spend their time answering our surveys and provide useful suggestions. We thank the anonymous reviewers for their constructive comments. This work was supported, in part, by the National Natural Science Foundation of China (61472175, 61472178, 91418202), the National Natural Science Foundation of Jiangsu Province (BK20130014), the China Scholarship Council Scholarship, DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, and 1320306, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub", Empirical Software Engineering, pp. 1–37, 2015.

[2] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories", in Procedings of 2009 IEEE 31st International Conference on Software Engineering, 2009, pp. 298–308.

[3] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows", in Proceedings of the 32nd International Conference on Software Engineering, 2010, pp. 495–504.

[4] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Not my bug! and other reasons for software bug report reassignments", in Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work, 2011, pp. 395–404.

[5] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users", in Proceedings of the 2010 ACM conference on Computer supported cooperative work, 2010, pp. 301–310.

[6] K. Blincoe, F. Harrison, and D. Damian, "Ecosystems in GitHub and a method for ecosystem identification using reference coupling", in Proceedings of the 12th Working Conference on Mining Software Repositories, 2015, pp. 202–207.

[7] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened", in Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 1074–1083.

[8] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD", in Procedings of 8th Working Conference on Mining Software Repositories, 2011, pp. 143–152.

[9] J. Bosch and P. M. Bosch-Sijtsema, "Softwares product lines, global development and ecosystems: collaboration in software engineering", in Collaborative Software Engineering, 2010, pp. 77–92.

[10] S. Jansen, A. Finkelstein, and S. Brinkkemper, "A sense of community: a research agenda for software ecosystems", in Procedings of 31st International Conference on Software Engineering - Companion Volume, 2009, pp. 187–190.

[11] D. G. Messerschmitt and C. Szyperski, Software Ecosystem: Understanding an Indispensable Technology and Industry, vol. 1. 2003.

[12] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes, "The small project observatory: visualizing software ecosystems", Science of Computer Programming, vol. 75, no. 4, pp. 264–275, 2010.

[13] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to API evolution? The Pharo ecosystem case", in Procedings of 2015 IEEE International Conference on Software Maintenance and Evolution, 2015, pp. 251–260.

[14] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem", in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, p. 56:1--56:11.

[15] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study", Empirical Software Engineering, vol. 20, no. 5, pp. 1275–1317, Oct. 2015.

[16] A. Decan, T. Mens, M. Claes, and P. Grosjean, "On the development and distribution of R packages: an empirical analysis of the R ecosystem", in Proceedings of the 2015 European Conference on Software Architecture Workshops, 2015, p. 41:1--41:6.

[17] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub meets CRAN: an analysis of inter-repository package dependency problems", in Procedings of International Conference on Software Analysis, Evolution, and Reengineering, 2016, pp. 493–504.

[18] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell, "Understanding “watchers” on GitHub", in Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 336–339.

[19] W. Wang, G. Poo-Caamano, E. Wilde, and D. M. German, "What is the gist? Understanding the use of public gists on GitHub", in Procedings of 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 314–323.

[20] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository", in Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, 2012, pp. 1277–1286.

[21] E. Kalliamvakou et al., "Open source-style collaborative development practices in commercial projects using GitHub", in Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 574–585.

[22] A. Lima, L. Rossi, and M. Musolesi, "Coding together at scale: GitHub as a collaborative social network", in Proceedings of the 8th International Conference on Weblogs and Social Media, ICWSM 2014, 2014, pp. 295–304.

[23] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub", in Proceedings of 36th International Conference on Software Engineering, 2014, pp. 356–366.

[24] J. Tsay, L. Dabbish, and J. Herbsleb, "Let’s talk about it: evaluating contributions through discussion in GitHub", in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 144–154.

[25] G. Gousios, A. Zaidman, M. A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator’s perspective", in Procedings of the 37th International Conference on Software Engineering, 2015, pp. 358–368.

[26] G. Georgios and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor’s perspective", in Proceedings of the 38th International Conference on Software Engineering, 2014, pp. 285–296.

[27] Github Inc., "GitHub Octoverse 2016", 2016. [Online]. Available: https://octoverse.github.com/.

[28] F. Perez, B. E. Granger, and J. D. Hunter, "Python: an ecosystem for scientific computing", Computing in Science & Engineering, vol. 13, no. 2, pp. 13–21, Mar. 2011.

[29] SciPy developers, "Topical software - SciPy.org". [Online]. Available: http://scipy.org/topical-software.html.

[30] T. Punter, M. Ciolkowski, B. Freimut, and I. John, "Conducting on-line surveys in software engineering", in Proceedings of 2003 International Symposium on Empirical Software Engineering, 2003, pp. 80–88.

[31] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?", in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, 2008, pp. 308–318.

[32] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?", in Proceeding of 28th international conference on Software engineering, 2006, pp. 361–370.

[33] T. T. Nguyen, T. N. Nguyen, E. Duesterwald, T. Klinger, and P. Santhanam, "Inferring developer expertise through defect analysis", in Proceedings of 34th International Conference on Software Engineering, 2012, pp. 1297–1300.

[34] M. Ohira, A. E. Hassan, N. Osawa, and K. Matsumoto, "The impact of bug management patterns on bug fixing: a case study of Eclipse projects", in Proceedings of 2012 28th IEEE International Conference on Software Maintenance, 2012, pp. 264–273.

[35] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes", in Proceedings of 35th International Conference on Software Engineering, 2013, pp. 332–341.

[36] C. Bogart, C. Kastner, and J. Herbsleb, "When it breaks, it breaks: how ecosystem developers reason about the stability of dependencies", in Procedings of 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop, 2015, pp. 86–89.

[37] Isaacs, "Add explicit `+1` feature for issues that isn't a comment", 2013. [Online]. Available: https://github.com/isaacs/github/issues/9.

[38] loureirorg, "Idea: A voting system for comments", 2014. [Online]. Available: https://github.com/isaacs/github/issues/209.

[39] R. Holmes and R. J. Walker, "Customized awareness: recommending relevant external change events", in Proceedings of the 32nd InternationalConference on Software Engineering, 2010, pp. 465–474.