Is Taking On Dependencies Becoming A Costly Affair?

Evaluating the Coordination Costs of Managing Upstream Dependencies

Anonymous Author(s)

ABSTRACT

Modern day software development is about building new software products fast and with fewer lines of code. This is often accomplished by taking on dependencies to reuse existing code in other software packages. However, developers often have no idea what long-term costs are associated with managing dependencies. Since getting work done in software projects often requires coordination among various stake-holders, we evaluate the impact of upstream dependencies in terms of the coordination costs they incur. We operationalize coordination costs as the time taken to complete, the number of messages exchanged, and the number of people involved in certain work-items of a project. We introduce the notion of external work-items as those work-items in which one or more upstream dependencies are involved. We use issues on GitHub's issue tracker as work-items and show using a large sample of repositories from PyPI and R ecosystems that, while over thirty percent of issues are external work-items, they account for more than fifty percent of the overall coordination costs. Moreover, we show that external work-items on average have higher coordination costs than other work-items even after controlling for possible confounds, thus indicating the presence of additional coordination in work-items involving upstream dependencies. Finally, we show that certain packages that go through a rigorous vetting process incur lower coordination costs, thereby also demonstrating the reliability of our measures.

KEYWORDS

Upstream dependencies; Coordination Costs; Software reuse; Dependency management; Software ecosystem; GitHub

ACM Reference format:
Anonymous Author(s). 2017. Is Taking On Dependencies Becoming A Costly Affair?. In *Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, 4–8 September, 2017 (ESEC/FSE 2017),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

ESEC/FSE 2017, Paderborn, Germany

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent studies have shown that not only has there been a significant growth in the number of software packages in popular ecosystems such as npm, R/CRAN and RubyGems but also a proportional increase in the number of dependencies among these packages [26]. Developers take on dependencies because it helps them reuse code that is well tested [38], and enables them to quickly integrate new functionality [14, 15, 33] even with a limited set of skills [53]. However, developers often have little or no idea what long-term costs are associated with managing upstream (or external) dependencies. In this paper, we propose an approach to assess these costs, and empirically evaluate its usefulness.

Several solutions to manage upstream dependencies in software projects exist. Developers adopt certain practices to improve communication between upstream and downstream projects [25], ecosystem level policies have been implemented to maintain compatibility among a set of packages [52], and there are commercially available tools such as Greenkeeper [9] and Gemnasium [7] to manage dependencies. However, several studies have shown that managing dependencies still continues to cause problems. A few examples include, build failures in Apache and Eclipse ecosystems caused by API incompatibilities [57], installation problems due to version conflicts among certain packages [23, 45], and migrating to a different dependency due to performance issues [50]. A more severe problem occurred in 2016 when a developer removed his package (left-pad) from the NPM repository. Since thousands of other packages had taken a dependency on this package, it left many developers including companies like Facebook unable to build and deploy their apps [56]. This example, although an extreme case, illustrates how fragile and complex the web of dependencies is in modern day software development.

Despite the breadth of study on dependency management in the software engineering literature [25, 32, 40, 42, 46], we know very little about the long-term maintenance costs of managing upstream dependencies. However, the development and maintenance effort involved in a software project can often be gleaned from work-items such as bug reports, code patches, discussions on mailing lists, etc. Completing a work-item, often requires coordination among various stake-holders [22], and these coordination efforts often make up a significant proportion of the overall development and maintenance effort [21, 22, 58]. Therefore, we analyze the long-term costs of managing dependencies in terms of the coordination costs involved in completing certain types of work items.

We hypothesize that work items in which one or more upstream dependencies are involved could reflect the coordination costs of managing those dependencies. We refer to these as *external* work-items and identify them using a

classification approach that is based on key-word filtering technique [31, 50] along with a few heuristics (explained in Sec. 3.2). For projects on GitHub, the issues reported on its issue tracker are common forms of work-items. On a sample of about 1.2 million issues from 27,420 GitHub repositories that are part of the PyPI ecosystem of packages, we identified about 37% of the issues as external work-items using our classification approach. Since frequencies of external work-items do not accurately reflect the impact of managing dependencies, we measure the coordination costs of resolving external work-items. Coordination costs of resolving a work-item (or issue) manifest in the form of delay (or time taken to resolve the issue) [35], size or the number of messages in related discussion threads [21, 51], and number of people involved [19, 35] in the issue. We found that the coordination costs of external work-items in our dataset accounted for at least 50 percent of the overall coordination costs across all measures. We also found that an external work-item on an average had higher coordination costs even after controlling for certain attributes of work-items and the projects they belong in. We ran our analysis on another set of 93,967 issues from 4036 repositories that are part of the R ecosystem of packages, and found comparable results, thus demonstrating the robustness of our results.

Software packages, however, differ widely in how they are developed, managed and used. Some packages for example, go through a rigorous vetting process that improves their maintainability. Such as packages that are distributed as part of the Python installation or packages accepted to be included in CRAN. However, it is not clear to what extent these vetting processes are useful in reducing project maintenance costs of managing dependencies. We found that external work-items that involve vetted packages take less time to close than external work-items that involve non-vetted packages, suggesting the difference in the coordination costs they incur.

Our key contributions in this research are as follows. First, we show that external work-items (or issues involving upstream dependencies) take up at least half of the overall project maintenance measured in terms of coordination costs. This along with our approach of identifying external work-items is useful for developers and managers in assessing the role of external dependencies in projects. Second, we show that issues that involve upstream dependencies incur significantly higher coordination costs irrespective of the type of issue and the nature of the project it belongs to. This shows that managing dependencies is still a cause of concern in software engineering, and suggests the need for further research in this direction. Third, we evaluate the usefulness of vetting processes through measures of coordination costs and show that packages that go through certain vetting processes incur lower coordination costs. This also shows that coordination costs can be a reliable measure to evaluate dependency management solutions and can also be useful for ecosystem administrators in making informed policy decisions.

2 BACKGROUND AND RESEARCH QUESTIONS

It is a common notion in software development to not reinvent the wheel. As a result, software developers reuse existing code often available in the form of libraries [41], packages [28] and other formats, which are broadly referred to as upstream (or external) dependencies. The benefits of software reuse [43] are quite evident as it helps developers quickly integrate new functionality [14, 15, 33] even with limited skills [53], and improves reliability [38] since code used and tested by many people may have fewer defects.

Taking on dependencies to reuse code is not a new phenomenon in software engineering, however, the last few years has seen an accelerated growth in the availability of reusable code (e.g. software packages) and the number of dependencies among them [26]. Moreover, social coding platforms like GitHub are making it easier for developers to search and integrate existing code [59], which further facilitates sharing and reuse of code. Therefore, dependency management is a more relevant problem to study now than ever before.

There are, however, a number of challenges with managing external dependencies [24, 25, 32]. Of these challenges, the most widely studied is change management: the process by which changes made to *upstream* dependencies affect *downstream* projects. Since change in most successful projects is inevitable [29], certain changes (or upgrades) could cause several problems to downstream projects. These include failed installs due to version conflicts [45], failed builds due to API incompatibilities [57], newly introduced bugs [25] and impacts on performance [50].

Apart from change management, downstream projects have to deal with a myriad of issues when managing upstream dependencies. For example, if a particular library becomes obsolete, migrating to a new library could become an overwhelming task [50]. Downstream projects also spend time asking questions about upstream dependencies on forums such as StackOverflow when upstream packages have insufficient API documentation [54]. Since there are several activities that software projects are involved in when it comes to managing upstream dependencies, it is important that we first understand the overall costs they incur. Since external work-items in our approach represent the activities that involve one or more upstream dependencies, we ask:

RQ1: What are the coordination costs of external work-items?

For certain types of work-items in a software project, the coordination costs can be measured in terms of a) the time taken to complete [35], b) the number of messages (or comments) in related discussion threads [21, 51], and c) the number of people [19, 35] involved in it. However, in order to make these measures of coordination costs more meaningful, we express it in terms of relative costs. One approach we take is to study these costs as proportion of overall project costs. This although is beneficial and we discuss its results in RQ1, it does not tell us the impact of an individual external work-item. Therefore, we measure the average coordination cost of an external work-item relative to the coordination

cost of an *internal* work-item. The class of work-items, which we refer to as *internal* represent the internal maintenance of a project that is usually within the purview of a project, and as a result are a reasonable baseline to compare external work-items against. Apart from intuition, there is however, little evidence that suggests that the coordination costs of external work-items must be higher than internal ones. While prior work has shown that dependencies across divisions in an organization and across geographical boundaries [35] incur higher coordination costs, open source software development differs in that it is almost entirely distributed. In other words, coordination involving either internal or external work-items mostly happens online, using the same communication medium, except the involvement of code and members of upstream dependencies in external work-items. Therefore, we ask:

RQ2: To what extent do external work-items incur higher or lower coordination costs compared to internal work-items?

Answering RQ2 is important for two reasons; first it allows us to quantify the cost of managing upstream dependencies irrespective of various attributes of a work-item or the project it belongs to. Second, RQ2 combined with RQ1 can tell us to what extent is managing dependencies a cause of concern in software engineering.

However, taking on dependencies often involves a careful decision making process. Developers look for certain qualities such as popularity, reliability, stability, etc. in packages before taking on dependencies. One of the ways packages acquire these qualities is when they are put through rigorous vetting processes usually determined by the policies of the ecosystem or certain developer practices. For example, PyPI packages that are distributed as part of the Python installation are known to comply with certain policies to improve maintainability. The CRAN (Comprehensive R Archive Network) ecosystem [31] for R packages requires that when making API breaking changes developers must coordinate with downstream projects before getting it released [20]. Although the underlying vetting processes and its usefulness might differ across ecosystems, external work items involving vetted packages should incur lower coordination costs than external work-items involving non-vetted packages. However, we do not know if a given vetting process moderates the coordination costs of the corresponding external work-items and to what extent. Therefore, we ask:

RQ3: Do external work items involving vetted packages incur higher or lower coordination costs compared to non-vetted packages?

The results of RQ3 can also be used determine whether coordination costs can reflect the usefulness of a given vetting process. As a result, helps assess the reliability of coordination costs as a measure for evaluating dependency management solutions and policies / practices.

Our research differs from previous studies in several ways. First, prior studies have looked at specific types of dependency management problems. For example, Claes et al. [23] studied the extent of Debian package conflicts and found that the fraction of strong conflicts remains constant over time. Wu et al. [57] found that in Apache and Eclipse ecosystems, missing classes and methods happened more frequently than missing interfaces with new framework releases. Decan et al. [28] found that about 41 percent of build errors on CRAN were due to dependency updates. Our research, however, differs in that we do not look at a specific type of dependency management problem rather we study the overall effect of dependencies on project maintenance.

3 METHODOLOGY

Our research questions and approach are independent of the type of software projects. However, since practices and tools for dependency management differ considerably across software ecosystems (and programming languages), we reduce irrelevant variation in the data by studying coordination costs in the context of projects from a single ecosystem. We begin with a case study of PyPI (Python Package Index) [10], and then replicate our study with projects from the R ecosystem [47] in order to validate the robustness of our results.

PyPI is a repository of software packages for the Python programming language, which has been heavily adopted by industry [11]and is also gaining popularity in academia [1, 12]. The PyPI ecosystem is mature and has stable packages for a variety of applications including web frameworks, scientific computing, and multimedia. R on the other hand, is also a well-established ecosystem with thousands of packages [27, 31]. It has had an accelerated growth in the last few years and was the fastest growing language on StackOverflow in 2015 [3]. However, R is primarily used in scientific computing, statistical and data analysis, is more widely used in academia than in industry, and has a bigger focus on reproducibility and replicability [2]. Since these two ecosystems differ in many ways, it helps to further support the generality of our findings. In this section, we first discuss our data collection and curation process. We then explain our classification technique to identify external work-items. Finally, we present the details of our statistical models.

3.1 Datasets

As of Feb. 2017, the PyPI repository has about 99,064 Python packages. We chose a sample of 44,465 packages that had their source code available on GitHub repositories. Among them, 27,420 repositories used GitHubs issue tracker, and we extracted 1,436,826 issues of which 678,937 were pull requests and 757,889 were bug reports created before Jan. 2017.

R package binaries are primarily distributed through CRAN (Comprehensive R Archive Network), with a few packages distributed through BioConductor [5]. GitHub, however, is emerging as a major platform for the development of R packages since R-forge has been on the decline [28]. Moreover, GitHub is also used in the distribution of R package binaries for packages that are not included in CRAN or BioConductor [28]. We therefore, identified all R package repositories on GitHub, first using GitHub Linguist [8] to identify R related repositories, and then checking for the presence of a package metadata file called DESCRIPTION [31] which is

characteristic of an R package. We found 19,233 R package repositories and after eliminating mirrored repositories [28] gave us 10734 repositories. Among them, 4,036 repositories had used GitHubs issue tracker, and we extracted 93,967 issues of which 25,491 were pull requests and 68,476 were bug reports created before Jan. 2017.

3.2 Classification of External Work-Items

We define external work-items as those activities in a software project that are caused by its upstream dependencies or arise as a result of managing them. Our approach to identify external work-items is based on a keyword based filtering technique previously used to identify whether a given mailing list discussion was related to a set of packages [31]. It is based on the premise that if the name of at least one upstream dependency is never mentioned in the discussions of a work-item then it cannot be an external work-item. For projects on GitHub, the issues reported on its issue tracker are common forms of work-items. We briefly list the steps involved in our approach below followed by more detailed explanations,

(1) Identify all upstream dependencies of a project (in this case, a GitHub repository).
(2) Exclude code blocks (text enclosed in triple backticks) from the contents (title, body, and comments) of issues.
(3) Mark issues in a project as external if the remaining content (after step 2) contains the name of at least one upstream dependency of the project.
(4) Discard issues marked as external in step 3 in which the names of upstream dependencies are all English dictionary words.

The process of identifying upstream dependencies of a project often depends on the programming language. For a Python project, one approach is to identify all packages mentioned in the import statements of all its source files. Since Python also allows importing modules from within a project, care must be taken to not include those as upstream dependencies. The process is similar for R projects except for differences in the syntax of import statements. In addition, R packages also contain a package metadata file, which often list a set of mandatory and optional dependencies of the package. We found that the nave keyword based filtering approach [31] had two major sources of error. First, when names of upstream dependencies appear inside a code snippet or error stack trace in the issue discussions. Fortunately, GitHub provides a standardized markdown [6] using triple backticks that users almost always use to annotate code blocks and error traces. Therefore, we filter out all text enclosed within triple backticks from the contents of issues before performing the classification.

The other major source of error is when upstream dependency names are used in the context of a regular English sentence in issue discussions. This occurs when the name of a package is an English dictionary word. In our dataset, we found 2710 upstream dependency names in PyPI were English dictionary words and 313 in R. Issues in which the names of upstream dependencies are all English dictionary words are ambiguous and cannot be conclusively classified as external or internal. Since the name of a package has no bearing on the way a package is developed or used, discarding these issues has very little risk in introducing any biases in our analysis of coordination costs. As we discuss in Sec. 4.2, treating these as external or internal did not have any major changes in the results of our models.

Although we found other sources of errors in our classification approach, they were not as frequent as the ones discussed above, and therefore, were not accounted for. A few examples include: a) Use of alternate names or acronyms of dependencies in issue discussions, b) code snippets and program output are not enclosed in triple backticks, c) issue discussions mention dependencies that have been removed from the project. These errors, although, are not strictly random, could occur in either misclassifying an external (false negative) or an internal issue (false positive), and since we use relative measures, as we discuss later, it may not have a significant effect on our analysis. However, we also performed a manual inspection of our classification approach on 500 issues (110 PRs and 390 bug reports) giving us an F-score of about 0.84 (precision and recall were 0.81 and 0.88 respectively), which is reasonable based on other text classification studies [30].

3.3 Coordination Costs

Managing dependencies requires coordination among various stake-holders [22], and related coordination activities can be gleaned from work-items such as bug reports and pull-requests on GitHubs issue tracker. The coordination cost of completing a work-item can be measured using a) time taken to complete it [35], b) number of people involved in completing the work-item [19, 35], and c) amount of discussion among the people involved in the work-item [21, 51]

Since the measures of coordination costs are only relevant for completed work-items, and since we use GitHub issues as forms of work-items, we only retain issues that are closed or merged. We also exclude issues where time taken to close is zero or negative, which occurred when issues were migrated from other platforms such as Google Code. These, however, accounted for less than 0.01 percent of the total issues in our datasets. Table 1 gives the summary statistics of our measures of coordination costs after applying our classification approach (Sec. 3.2) on the datasets. Table 1 also gives the descriptives for *external* and *internal* issues.

In order to test for a difference between the median coordination cost measures for external and internal issues, we use the non-parametric Wilcoxon Rank-Sum test [55]. We use the wilcox.test function as part of the stats package in R, which uses the Hodges-Lehmann estimator [36] to compute the effect size and its confidence interval.

3.4 Linear Mixed-Effects Model

The Wilcoxon Rank-Sum test does not consider any confounds when comparing the coordination costs of external and internal issues. Therefore, we use a multiple regression technique to control for confounding effects such as whether

Table 1: Descriptives of Coordination Cost Measures

	PyPI		R	
Coordination Cost Measure	**Mean (St. Dev.)**	**Median**	**Mean (St. Dev.)**	**Median**
Time (in days)	47.81 (180.1)	1.07	46.56 (128.68)	1.67
Time (External)	66.24 (235.49)	2.77	53.76 (134.23)	3.79
Time (Internal)	37.06 (136.72	0.76	42.93 (125.63)	0.98
Comments	2.69 (5.11)	1	2.13 (3.95)	1
Comments (Ext.)	4.61 (7.32)	3	3.92 (5.77)	2
Comments (Int.)	1.56 (2.58)	1	1.23 (2.06)	1
Users	1.94 (1.30)	2	1.64 (0.86)	1
Users (Ext.)	2.37 (1.66)	2	2.08 (1.04)	2
Users (Int.)	1.68 (0.94)	2	1.42 (0.64)	1
Rows	1,017,210		69,626	
Rows (Ext.)	374,743		23,367	
Rows (Int.)	642,467		46,259	

the issue is a pull-request or a bug report, is a feature request, contributor status of the submitter [51], among others, which we discuss below. The issues, however, are not strictly independent data points since multiple issues could belong to a single repository. As a result, we use a linear mixed-effects model with a random effects term for repository. We implement our models using using the lmerTest [39] package in R, which is built on top of the lme4 package [18] but provides p-values based on Satterthwaites approximations [48]. The effect size of coefficients is computed using the ANOVA analysis. We found no evidence for multi-collinearity as inferred from the VIF implementation [13] for mixed models adapted from the rms package [34] in R. We compute the marginal (R^2_m) and conditional (R^2_c) coefficient of determination as implemented in the MuMIn package [17, 44] to assess the fit of our model. We also found no violation of the normality assumption as observed from our probability (or Q-Q) plots.

1) Outcome Variables: Our goal is to use our models to evaluate the difference in measures of coordination costs between external and internal issues. Previous research, however, has shown that delay in software development activities increases with the number of people involved [35]. A similar argument could be made for the number of comments in an issue [51]. Moreover, the number of unique users involved in an issue discussion could also control for the level of interest in that issue. Therefore, we build two models, one with time taken to close an issue and another with number of comments in an issue as the outcome variable, and we use the number of users (identified by their unique GitHub login) involved in an issue as a control variable. We fit the models to the PyPI and R datasets separately.

2) Covariates: Our main predictor is the dichotomous variable Is external, which takes the value, true for issues classified as external, and false otherwise. We also include a random slope term for Is external to vary across repositories, as not accounting for the variability in the coefficient of predictors across different groups could result in Type I errors [16, 49]. We list below the other covariates in our models:

a) Is pullrequest: This is a dichotomous variable indicating whether the issue is a pull-request or not. Although both Pull requests and bug reports are managed in GitHubs issue tracker, they are designed for different purposes.

b) Is feature: This is a dichotomous variable indicating whether label names assigned to an issue contain either enhancement or feature. We decided not to filter out feature requests because they can also be considered as work-items in a project, and moreover, external issues such as adding dependencies, upgrading versions were often tagged as feature requests.

c) Number of cross-references: An issue can contain one or more cross-references to other issues specified using GitHubs Autolinked references (part of Flavored Markdown) [4]. Cross-references could imply that the issues are inter-dependent, and as a result, could have confounding effects on our results. Therefore, we include the number of cross-references in an issue as a control variable.

d) Is contributor submitted: This takes the value, true for issues that are reported or submitted by a contributor. We identify contributors of a repository as users with at least one authored commit in the repository.

e) #Dependencies: The number of unique upstream dependency names occurring in an issue. Since this variable is zero for all internal issues, it does not make sense to include in our model comparing external and internal issues. However, we use this in the analysis of vetted packages, which we discuss later in this section. Therefore, in Table 1 we show its summary statistics computed on only external issues.

f) Number of stargazers: We control for the popularity of a repository using the number of stars it has received [51].

g) Age of a Repository (years): The age is often an indication of maturity and is a widely used repository level control measure [51].

h) Size: The size of a repository can be measured in many ways. We use the number of contributors and the total size of all files in the repository as proxies for size. The latter is also often correlated with the number of lines of code. The repository level variables, number of forks and subscribers were found to be collinear with the number of stars for both the PyPI and R datasets, corroborating previous findings [51]. Table 2 shows the summary statistics of our independent variables. The dichotomous variables have been converted to numeric for illustrative purpose.

All variables have a high variance compared to their mean, therefore we log transformed all our continuous variables including the outcome variables to ensure normality and account for the high dispersion. Although the skewed variables are often a result of outliers, we, however, did not remove them, as it did not have any significant effect on the fit and results of our models. We centered all our continuous independent variables in order to be able to meaningfully interpret the intercept. We used *lsmeans* package in R to compute confidence intervals, which is based on the Satterthwaite degrees of freedom method [48].

3) Interaction terms: Pull requests and bug reports are two different types of work-items often used for different purposes, and therefore, it is plausible that it could have a moderating effect on the relationship between coordination cost measures and other independent variables, specifically if

Table 2: Descriptives of Independent Variables

	PyPI		R	
Coordination Cost Measure	Mean (St. Dev.)	Median	Mean (St. Dev.)	Median
Is External*	0.37 (0.48)	0	0.34 (0.47)	0
Is Pullrequest*	0.55 (0.5)	1	0.33 (0.47)	0
Is Feature*	0.08 (0.26)	0	0.1 (0.3)	0
#Cross-Refs	0.12 (1.01)	0	0.08 (0.31)	0
Is Contributor Submitted*	0.73 (0.44)	1	0.74 (0.44)	1
#Dependencies (Ext.)	2.46 (2.3)	2	1.75 (1.64)	1
#Stars (Repo)	109.72 (691.17)	8	20.04 (91.81)	3
Age in years (Repo)	3.08 (1.8)	2.84	2.72 (1.23)	2.44
#Contributors (Repo)	7.06 (19.96)	3	2.93 (4.7)	2
Size in KB (Repo)	4587.8 (68625.4)	234	15044.5 (81133.7)	786
Total Issues	1,017,210		69,626	
Total Repos	23,986		3309	

*Dichotomous variable (numeric coded as 0 = False, 1 = True)

the issue was reported by a contributor [51]. We therefore, test for a two-way interaction between the variables, *Is pullrequest* and *Is contributor submitted*. Similarly, we test for two-way interactions between our main predictor variable (*Is external*) and each of *Is pullrequest* and *Is contributor submitted*.

4) Analyzing Vetted Packages: For this analysis, we retain only external issues in our model and compare the difference in coordination costs of external work-items involving vetted and other packages. We introduce a new categorical predictor variable in our model *Depends on vetted*. This predictor is assigned YES if an external issue involves only vetted packages, NO if it involves only by packages that are not vetted, and BOTH if the issue involves a combination of both vetted and non-vetted packages. We recall that external issues are identified based on our classification approach described in Sec. 3.2, which involves checking for the occurrence of names of package dependencies in issue discussions. As a result, we also control for the number of unique upstream dependency names occurring in an issue.

For the purpose of this analysis, it is important that we identify a group of packages in a given ecosystem that more or less go through a similar vetting process. Therefore, we use the following criterion for selecting vetted packages in our datasets: a) In the PyPI dataset, packages that come bundled with the Python installation. We considered Python versions up to 3.4 and found 491 packages that met this criterion. As a result, the proportion of external issues in the three categories are: YES (28.3%), NO (56.6%), BOTH (15.1%). b) In the R dataset, packages that are included in the CRAN repository. As of Jan. 2017, there are 10,128 packages on CRAN.The proportion of external issues in the three categories are: YES (77.9%), NO (16%), BOTH (6.1%).

4 RESULTS

We present our findings in four parts. We first discuss the frequency of occurrence of external issues and their coordination costs (RQ1). Second, present our results of comparing the time taken to close external and internal issues (RQ2) for both ecosystem datasets. Third, along the same lines we present our results comparing number of comments (RQ2).

Table 3: Extent of External Issues

	PyPI		R	
	Bug-reports	Pull-requests	Bug-reports	Pull-requests
Open	65,202 (41.6%)	10,385 (40%)	5301 (31.1%)	328 (30%)
Closed	202,231 (44.2%)	172,512 (30.8%)	18,453 (39.8%)	4914 (21.1%)
Both	267,433 (43.6%)	182,897 (31.2%)	23,754 (37.4%)	5242 (21.5%)
Total	450,330 (37.5%)		28,996 (33%)	

Finally, we discuss our results of analyzing the usefulness of certain vetting processes in PyPI and R (RQ3).

4.1 RQ1: Coordination Costs of External Work-items

Before we study the coordination costs of external work-items and interpret the results, it is important that we know to what extent do they occur in our datasets. We discussed earlier that our classification approach is uncertain in classifying certain issues (dependency names with English dictionary words) into either external or internal. The total number of such issues we found were 232,420 (16.2%) in PyPI and 5641(6%) in R. Although we discarded these issues from our analysis, we found no significant effect in the results of our models, which we discuss in subsequent sections. In Table 3, we present the proportion of external issues classified by our approach.

Each cell in Table 3 represents the number and percentage of external issues for that particular category. For example, the first cell reads that among open bug reports in the PyPI dataset, 65,202 were classified as external and this accounted for about 41.6% of the total open bug reports in that dataset. The overall proportion of external issues is comparable in both ecosystems, 37.5% in PyPI and 33% in R, suggests that work-items related to external dependencies are fairly common irrespective of the programming language.

Between bug reports and pull requests, the fraction of external issues is consistently higher for bug reports across both the ecosystems, and the difference is substantial (about 12%, 16% higher in PyPI and R respectively). ***This suggests that external issues manifest more frequently as maintenance related work-items, indicating that projects have to deal with more undesirable forms of work-items when it comes to managing dependencies.*** Feature requests were not excluded as they were a small fraction and did create any significant differences.

The frequency, however, does not tell us about the coordination effort involved in completing external work-items. Therefore, we compute the proportion of coordination costs of external work-items over all work-items as shown in Table 4. We only use closed issues in Table 4 since by definition, coordination costs are only applicable for completed work-items.

Table 4: Coordination Costs of External Issues as a Percentage of Total Coordination Costs

	PyPI			R		
	Bug-reports	PRs	Both	Bug-reports	PRs	Both
Time	52%	47.2%	51%	38.6%	40%	39%
Comments	66.3%	60%	63%	63.6%	54.8%	62%
Users	52.6%	38.6%	45%	48.7%	28.1%	43%

Comparing Table 3 and Table 4 we find that, even if the fraction of external issues are less than half the total issues, their coordination costs account for more than half the overall costs (with comments as a measure, and time for PyPI). The results are mostly comparable across PyPI and R, with the exception of time taken, which is considerably higher for PyPI. This difference is interesting because the mean time to close an issue is almost same (Table 1) between PyPI and R, but the mean time to close external issues is noticeably higher in PyPI than in R. One possible explanation for this could be that a most of widely used packages in R are on CRAN, which enforces certain policies for coordination between upstream and downstream packages [20], and thereby reducing certain forms of coordination costs. We will analyze this is more depth when we answer RQ3 in Sec. 4.4.

As we have seen, coordination cost measures are more meaningful when we can express them in terms of relative costs. One useful approach, which we discuss in the following section is by comparing external and internal issues, and this also gives us more significant insights and understanding of coordination costs.

4.2 RQ2: Comparing External and Internal Work-items: Time Taken to Close

The Wilcoxon rank-sum test revealed that the difference in median time to close between external and internal issues is statistically significant ($p < 2.2e^{-16}$) with an effect size of 0.489 days and a 95% confidence interval of [0.475, 0.5] in the PyPI dataset. The effect size in the R dataset was slightly higher at 0.56 days with a confidence interval of [0.51, 0.62]. Although this shows that external issues do in fact take longer to close compared to internal issues, it does consider other possible confounds. Therefore, we fit a linear mixed effects regression model to our data with time taken to close an issue as the response variable. Table 5 shows the fixed effect coefficients (and corresponding standard errors) of the model run separately PyPI and R. In order to avoid errors due to floating-point precision, we use time units as seconds in all our models.

In PyPI, our main predictor, Is External is statistically significant with a coefficient of 0.18. Interpreting this, however, is more intricate due to the presence of interaction terms. The result shows that among bug reports submitted by non-contributors, external ones take 19.7% ($e^{0.18}1$) more time to close than internal ones, with all other variables kept constant. However, among bug reports submitted by contributors, external ones take about 12.2% ($e^{0.18-0.31}1$) less time

Table 5: Fixed Effect Coefficients with Response as Log(Time to close in secs)

	PyPI		R	
	Coeff. (Std. err.)	Sum Sq.	Coeff (Std. err.)	Sum Sq.
(Intercept)	1.20^{***} (0.032)		1.14^{***} (0.083)	
#Users (log)	3.66^{***} (0.012)	898408	3.16^{***} (0.057)	29706
Is External	0.18^{***} **(0.019)**	**10378**	**0.10 (0.065)**	**2189.6**
Is Pull-request	-0.82^{***} (0.017)	229752	-0.70^{***} (0.091)	12065
Is Feature	1.44^{***} (0.013)	117316	1.51^{***} (0.045)	10794
Is Contributor Submitted	1.29^{***} (0.013)	332	1.67^{***} (0.052)	269
#Cross-Refs (log)	0.51^{***} (0.014)	13191	0.44^{***} (0.067)	407.8
#Stars (Repo)(log)	0.08^{***} (0.008)	767	0.18^{***} (0.03)	351.2
Age in years (Repo)(log)	0.93^{***} (0.025)	12883	0.79^{***} (0.11)	486
#Contributors (Repo)(log)	0.12^{***} (0.018)	393	-0.06 (0.08)	5.4
Size in KB (Repo)(log)	-0.06^{***} (0.007)	603	-0.02 (0.02)	15.1
Is External * Is Pull-request	0.98^{***} (0.016))	37601	1.58^{***} (0.068)	5196.3
Is External * Is Contrib. Submi.	-0.31^{***} (0.017)	3086	-0.21^{**} (0.067)	97
Is Pull-request * Is Contr. Submi.	-0.22^{***} (0.017)	147640	-3.63^{***} (0.093)	14569
AIC / BIC / Log. Lik.	5207682 / 5207895 / -2603823		359375.9 / 359540 / -179669.9	
R^2_m / R^2_c	0.25 / 0.4		0.29 / 0.45	
#Observations:	1,017,210		69,626	
#Groups: (repos)	23,986		3309	

to close than internal ones. This might seem unusual but we see similar results with the R dataset as well (statistically significant with about 10.5% increase and 11.6% decrease in the above two cases respectively). The reasons behind this are not entirely evident, however, we found that the ratio of external to internal bug reports submitted by contributors was considerably lower (0.56 for PyPI, and 0.42 for R) compared to that of non-contributors (1.2 for PyPI, and 1.4 for R). This could mean that among contributor submitted bug reports there are far fewer external ones to deal with than among non-contributor submitted ones.

Among pull requests, however, the story is quite different. Among the non-contributor submitted pull requests, external ones do take longer than internal ones but the difference is considerably higher at over twice ($e^{0.18+0.98}1$) longer in PyPI and over four times ($e^{0.1+1.58}1$) longer in R. Since these numbers seem considerably higher that what we would expect, we cross verified with the corresponding mean and median values. We found that non-contributor reported external pull requests had a mean and median time of 71.3 and 3.98 days respectively compared to 48 and 0.84 days for internal ones respectively in PyPI. The mean and median time for external pull requests in R were 74 and 7.76 days compared to 44.26 and 0.87 days for internal ones respectively. This although, corroborates with the results of our models, they may not be straightforward to interpret. This is because in most cases a user whose pull request is merged becomes a contributor in that project. Therefore, non-contributor submitted pull

requests are rare and might occur say, when a pull request is rejected. The total non-contributor submitted pull-requests we found in our datasets were 59,770 in PyPI and 1642 in R. Although this is a small fraction of issues and might be difficult to interpret, it is important that we account for it in the model.

Among contributor submitted pull-requests, the model shows a huge difference in the time taken to close between external and internal ones. It takes 1.34 times ($e^{0.18+0.98-0.31}1$) longer in PyPI and 3.35 times ($e^{0.1+1.58-0.21}1$) longer in R to close external pull requests. This although seems significantly higher than what we would expect, the results we see is primarily because pull-requests in general take much shorter to close, and so even a small difference shows up as a large relative difference. For example, the mean time to close contributor submitted internal pull requests is 9.36 and 7.48 days in the PyPI and R datasets respectively. The same for external ones is 19.38 and 17.98 days respectively, which corroborates with the substantial relative increase that the model indicates. Nevertheless, this strongly suggests that external pull-requests incur higher coordination costs measured in terms of the time taken to close. Therefore, it is possible that external pull requests might either require more inspection or may have difficulties finding appropriate reviewers.

We evaluate the robustness of our results in several ways. First, the results from the two independent datasets - PyPI and R are fairly comparable in terms of the polarity of the coefficients and their effect sizes. Second, we show that the model fits the data reasonably well with 40% and 45% of the variability explained when including both fixed and random effects (R2c) for the PYPI and R datasets respectively. The variance of our random intercept, slope and the correlation between the two were 2.5, 0.9, -0.53 respectively for PyPI and 3.2, 0.8, -0.7 respectively for R. The high negative correlation indicates that as the baseline time (intercept) increases, the difference in time between external and internal issues decreases. Finally, we evaluate the sensitivity of our model to the set of issues that were discarded as a result of our classification approach (Sec. 3.2). We compared our results with models that included the discarded set as either internal or external, and found no significant differences. We also found our model results to be generally insensitive to outliers.

Due to the presence of interaction terms in our model, we interpreted the difference in time taken between external and internal issues among four different categories of issues. Overall, we found that external issues do take longer, except for bug reports submitted by contributors of a project. We also found no overlap in the 95% confidence intervals for the average predictions (log(time) in this case) between external and internal issues ([12.9, 13.0] and [12.38, 12.5] for PyPI, and [12.35, 12.63] and [11.56, 11.86] for R). Since we control for several variables, such as the number of unique users to account for level of interest, the number of cross references to account for complexity, and other variables to account for the popularity, maturity, and size of a repository, our claims on the difference in time taken between external and internal issues is well substantiated and has implications for the broader software engineering literature.

Table 6: Fixed Effect Coefficients with Response as Log(Number of Comments)

	PyPI		R	
	Coeff. (Std. err.)	**Sum Sq.**	**Coeff (Std. err.)**	**Sum Sq.**
(Intercept)	0.89*** (0.003)		0.74*** (0.008)	
#Users (log)	1.84*** (0.002)	228216	1.98*** (0.008)	11905
Is External	**0.18*** (0.003)**	**1555**	**0.16*** (0.009)**	**170**
Is Pull-request	-0.11*** (0.002)	1145	-0.13*** (0.012)	70.4
Is Feature	0.06*** (0.002)	199	0.06*** (0.006)	19.2
Is Contributor Submitted	0.02*** (0.002)	34	0.07*** (0.007)	5.4
#Cross-Refs (log)	0.22*** (0.002)	2481	0.29*** (0.009)	177
#Stars (Repo)(log)	0.01*** (0.0008)	25	0.009** (0.003)	1.7
Age in years (Repo)(log)	-0.001 (0.003)	0	-0.005 (0.01)	0
#Contributors (Repo)(log)	-0.007*** (0.002)	3	-0.005 (0.007)	0.1
Size in KB (Repo)(log)	0.006*** (0.0007)	12	-0.001 (0.002)	0.1
Is External * Is Pull-request	0.05*** (0.002)	108	0.11*** (0.009)	23.4
Is External * Is Contrib. Submi.	0.02*** (0.002)	13	-0.03** (0.009)	1.7
Is Pull-request * Is Contrib. Submit.	-0.03*** (0.002)	33	-0.10*** (0.01)	12.8
AIC / BIC / Log. Lik.	1164586.4/ 1164799.4/ -582275.2		78322.5 / 78487.3 / -39143.3	
R^2_m / R^2_c	0.67 / 0.71		0.66 / 0.70	
#Observations:	1,017,210		69,626	
#Groups: (repos)	23,986		3309	

4.3 RQ2: Comparing External and Internal Work-items: Number of Comments

The Wilcoxon rank-sum test revealed that the difference in the median number of comments between external and internal issues is statistically significant ($p < 2.2e^{-16}$) with an effect size of 1.00 comments and a 95% confidence interval of [1.0001, 1.0002] in the PyPI dataset. The effect size in the R dataset was the same at 1.00 comments with a larger confidence interval of [1.00, 1.99]. As we did with our coordination cost measure, time, we fit a linear mixed effects model with number of comments as the response variable.

The coefficient of our main predictor as inferred from Table 6 is statistically significant and has a value of at 0.18 and 0.16 for PyPI and R respectively. This indicates the relative difference (about 20% and 17%) in the number of comments between external and internal bug reports submitted by non-contributors of a project. For contributor submitted bug reports the relative difference is still positive at 22% and 14% for PyPI and R respectively, which is unlike what we saw in our model predicting time. In other words, even though contributor submitted external bug reports take less time to close than internal ones, they however, involve more

comments indicating the higher coordination costs of external work-items.

With pull requests, the model shows a similar trend but with a higher relative difference. Among non-contributor submitted pull-requests, the relative difference in comments between external and internal ones is about 26% and 31% for PyPI and R respectively. This is also quite similar for contributor submitted pull requests with a relative increase of about 28% and 27% for PyPI and R respectively.

The model fits the data considerably better than our model predicting time. The model explains about 71% and 70% of the variability when including both fixed and random effects (R2c) for the PYPI and R datasets respectively. However, the random effect terms only explain a small amount of the variance with the residual variance being less than 0.2 for both PyPI and R. This is also because the number of users explains most of the variance as it is highly correlated with the response variable. Removing the number of users from the model did not affect the polarity of our coefficients, however, retaining it makes our claims even stronger that external issues involve more discussions than internal ones even when the number of users is kept constant. We also found no overlap in the 95% confidence intervals for the average predictions between external and internal issues ([1.07, 1.09] and [0.86, 0.87] for PyPI, and [0.92, 0.96] and [0.7, 0.73] for R), further supporting our findings.

The key insight based on both models is that coordination costs of external work-items are consistently higher than internal work-items. Since the reasons behind these results are not entirely intuitive, our results can spark further research in investigating these differences.

4.4 RQ3: Coordination Costs of External work-items Involving Vetted Packages vs Non-Vetted Packages

Table 7 shows the results of our models for comparing the coordination costs of external issues that depend on vetted package, non-vetted packages and both. Due to restrictions on space, we do not show the standard error and sum of squares measures in Table 7, which will be archived and made available along with our datasets[1].

With time as the response variable, the main effects of our predictor (Depends on Vetted) show a distinct difference between the levels YES and NO (relative to the baseline level BOTH). The difference is also more pronounced among contributor submitted bug reports. Among pull requests, however, there is no noticeable difference between the two levels. The results are not statistically significant in R to be able to interpret convincingly, however, we found very strong effects in PyPI with a less than fifty percent (about 32%) overlap in the 95% confidence interval of the overall average predictions YES and NO.

In the Comments model, the main predictor although is statistically significant in both PyPI and R, the difference between the two levels is however, negligible (with overlapping confidence intervals). In order to evaluate it with more degrees of freedom, we also included the internal issues in our model (the predictor coded with an additional category in this case). We, however, did not find any apparent differences in the results besides complicating the model.

Overall, it appears that taking dependencies on non-vetted packages is more likely to incur higher coordination costs than vetted packages, the difference however, is more noticeable with time as a measure than number of comments. It is also important to note that the difference is higher for bug reports than pull-requests, which suggests that the effect of non-vetted packages on coordination costs could be higher for maintenance related work-items. The baseline level, BOTH of our predictor also presents an interesting case. Even though we controlled for the number of dependencies involved in an issue, we found that the time taken for work-items that involve both types of dependencies take slightly less time to close on average than those that only involve only vetted packages. The difference although not conclusive (overlapping confidence intervals) suggests that vetted and non-vetted packages are used and occur in more complex ways that requires further investigation to understand the phenomenon better.

Table 7: Fixed Effect Coefficients

	Response: Log(time)		Response Log(#Comments)	
	Coeff. (Std. err)		Coeff. (Std. err)	
	PyPI	R	PyPI	R
(Intercept)	0.12***	0.12***	1.22***	1.16***
#Users (log)	3.53***	2.57***	1.63***	1.88***
Depends on Vetted: NO	0.36***	0.19	0.12***	0.07*
Depends on Vetted: YES	0.21***	0.10	0.12***	0.04
Is Pull-request	0.60***	0.40	0.08***	0.07
Is Feature	1.51***	1.65***	0.08***	0.13***
Is Contributor Submitted	0.75***	1.64***	0.13***	0.22***
#Cross-Refs (log)	0.38***	0.46***	0.16***	0.23***
#Dependencies (log)	0.45***	0.25***	0.50***	0.34***
#Stars (Repo)(log)	0.04***	0.12**	0.02***	0.03***
Age in years (Repo)(log)	0.77***	0.80***	-0.02***	-0.04*
#Contributors (Repo)(log)	-0.003	-0.16*	-0.03***	-0.04**
Size (KB) (Repo)(log)	-0.04***	-0.05*	0.01***	-0.002
Depends on Vetted (NO) * Is Pull-request	-0.72***	-0.63*	-0.10***	-0.17***
Depends on Vetted (YES) * Is Pull-request	-0.55***	0.29	-0.12***	-0.05
Depends on Vetted (NO) * Is Contrib. Sub.	0.20***	-0.09	-0.09***	-0.12**
Depends on Vetted (YES) * Is Contrib. Sub.	0.13***	-0.45*	-0.07***	-0.11***
Is Pull-request * Is Contrib. Submit.	-1.86***	-3.1***	-0.03***	-0.12***
AIC & BIC (rounded)	1.88E6	1.18E5	5.24E5	3.40E4
R^2_m / R^2_c	0.20 / 0.34	0.15 / 0.28	0.64 / 0.69	0.56 / 0.61
#Observations:	374743	23367	374743	23367
#Groups (repos):	14,586	1,797	14,586	1,797

5 DISCUSSIONS

The primary goal of our research was to evaluate the impact of upstream dependencies on software projects. We addressed our goal in three phases, first, we showed that external work-items in a project, i.e. those that involve upstream dependencies accounted for more than fifty percent

[1] https://github.com/arunk054/coordination-costs-fse17

of the overall coordination costs. Second, we showed that an external work-item on average has higher coordination costs than an internal work-item even after controlling for possible confounds. Third, we showed using coordination costs that there is value in the processes that go behind vetting certain packages. These findings put together also demonstrates that our method of identifying and evaluating coordination costs of external work-items is a reliable approach to assess the impact of upstream dependencies.

5.1 Implications for practice

In todays world of fast paced software development, the major challenges facing open-source software projects is being productive and at the same time being self-sustained. Achieving this, however, is not feasible without reusing code from other projects and thereby taking on dependencies, which as we showed comes at a cost. Therefore, both project managers and developers can benefit from tracking certain costs associated with managing dependencies. Our approach of measuring coordination costs to systematically assess the impact of upstream dependencies can be readily applied to almost all projects that use GitHubs issue trackers as their primary tool to coordinate work. For projects that are not on GitHub or do not use the issue tracker, our approach can be adapted to deal with other sources of work-items. However, when certain dependencies become too difficult to manage, developers often decide to either migrate to another library or completely do away with it by cloning or re-implementing the required functionalities. Our research can be extended to support such decisions, e.g. by comparing problematic package(s) with other packages as we did in Sec. 4.4 by comparing vetted and non-vetted packages.

Software projects, however, dont exist in isolation as they are becoming more and more inter-connected, and being present as part of a large ecosystem. Therefore, ecosystem policies have a critical role in facilitating reuse and coordination across projects. We showed that our approach can be extended to evaluate the coordination costs incurred by a subset of packages within an ecosystem. These packages for example built-in packages within the PyPI ecosystem and CRAN packages within the R ecosystem, are governed by certain policies and practices. Although we did not evaluate the effectiveness of a single policy, it is possible to adapt our approach to perform a longitudinal study where coordination costs can be compared before and after a particular policy is enforced. This could also give administrators a powerful tool to experiment with different policy decisions.

5.2 Implications for software engineering research

The literature on global software development has discussed several challenges and solutions associated with cross-site coordination [35]. However, as software projects are becoming increasingly inter-connected by reusing each others code, new solutions, be it tools, practices or policies are being constantly implemented to manage dependencies better. Yet, without understanding the longer-term impacts of this form of development, our solutions may not be as effective. Although our work showed the impact as measured in terms of coordination costs, there are several other forms costs that this could be modeled on. Our contributions therefore, could spark concerted effort in advancing research addressing this particular problem.

The notion of external work-items that we introduced could facilitate further research in understanding the various kinds of activities related to dependency management. It is however, not entirely apparent, why external work-items incur significantly higher coordination costs than internal ones. Since the involvement of upstream dependencies is the only way external work-items differ conceptually, it highlights the fact there are certain additional coordination activities that arise as a result of taking on dependencies. While further investigation on the reasons behind these differences is required, our approach and findings can make it easier to conduct focused studies in future.

5.3 Limitations

We filtered out repositories from our dataset that did not use the issue tracker. Since previous research has shown that many active projects on GitHub use an external issue database [37], there is a potential risk that this filtering process could introduce certain biases. Moreover, issues may not necessarily represent all forms of work-items in software projects. Our analysis was centered around measuring coordination costs of work-items. We adapted our approach (RQ3 in Sec: 4.4) to evaluate the coordination costs incurred from using a given set of packages. This although helps answer our research question, it is not appropriate if we are required to know in absolute measures, the coordination costs incurred by one or more package(s) and based on how widely it is used. For these purposes, coordination costs must be analyzed and aggregated on a per package basis.

6 CONCLUSIONS

External dependencies are the glue that holds together the globally distributed collaborations of software ecosystems, and so we sought to understand just how much development and maintenance effort goes into inter-project coordination. We showed that work-items involving external dependencies accounted for more than half of overall coordination costs. We fit our data to linear mixed effects models predicting coordination costs and found that on average coordination costs of external work-items were significantly higher than other work-items. Our findings show that managing dependencies is still a cause of concern in software engineering, and suggests the need for further research in this direction. There is scope for future work to improve the accuracy of our classification approach, and identify more fine-grained predictors of external problems. Our research has the potential to help developers and managers periodically evaluate their dependency management solutions, inform the design of ecosystem policies, and help developers more clearly see their place in their ecosystems large-scale collaboration.

REFERENCES

[1] 2014. Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities — blog@CACM — Communications of the ACM. (2014). (Accessed on 02/28/2017).

[2] 2015. CRAN Task View: Reproducible Research. https://cran.r-project.org/web/views/ReproducibleResearch.html. (2015). (Accessed on 02/28/2017).

[3] 2015. R is the fastest-growing language on StackOverflow. http://blog.revolutionanalytics.com/2015/12/r-is-the-fastest-growing-language-on-stackoverflow.html. (2015). (Accessed on 02/28/2017).

[4] 2017. Autolinked references and URLs. (2017). https://help.github.com/articles/autolinked-references-and-urls/

[5] 2017. Bioconductor - Home. https://www.bioconductor.org/. (2017). (Accessed on 02/28/2017).

[6] 2017. Creating and highlighting code blocks. (2017). https://help.github.com/articles/creating-and-highlighting-code-blocks/

[7] 2017. Gemnasium. (2017). https://gemnasium.com/

[8] 2017. Github Linguist. (2017). https://github.com/github/linguist

[9] 2017. Greenkeeper. (2017). https://greenkeeper.io/

[10] 2017. PyPI - the Python Package Index : Python Package Index. https://pypi.python.org/pypi. (2017). (Accessed on 02/28/2017).

[11] 2017. Python as a First Language. http://mcsp.wartburg.edu/zelle/python/python-first.html. (2017). (Accessed on 02/28/2017).

[12] 2017. Python scales new heights in language popularity — InfoWorld. http://www.infoworld.com/article/3012442/application-development/python-scales-new-heights-in-language-popularity.html. (2017). (Accessed on 02/28/2017).

[13] 2017. VIF Mixed Effect Models. (2017). https://github.com/aufrank/R-hacks/blob/master/mer-utils.R

[14] Rajiv D Banker and Robert J Kauffman. 1991. Reuse and Productivity in Integrated Computer-aided Software Engineering: An Empirical Study. *MIS Q.* 15, 3 (oct 1991), 375–401. DOI: http://dx.doi.org/10.2307/249649

[15] B Barnes, T Durek, J Gaffney, and A Pyster. 1988. Software Reuse: Emerging Technology. IEEE Computer Society Press, Los Alamitos, CA, USA, Chapter A Framewor, 77–88. http://dl.acm.org/citation.cfm?id=63179.67565

[16] Dale J Barr. 2013. Random effects structure for testing interactions in linear mixed-effects models. (2013). DOI:http://dx.doi.org/10.3389/fpsyg.2013.00328

[17] Kamil Bartoń. 2016. *MuMIn: Multi-Model Inference*. https://cran.r-project.org/package=MuMIn

[18] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using {lme4}. *Journal of Statistical Software* 67, 1 (2015), 1–48. DOI:http://dx.doi.org/10.18637/jss.v067.i01

[19] Kelly Blincoe, Francis Harrison, and Daniela Damian. 2015. Ecosystems in GitHub and a Method for Ecosystem Identification Using Reference Coupling. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 202–207. http://dl.acm.org/citation.cfm?id=2820518.2820544

[20] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM Press, New York, NY. http://breakingapis.org

[21] Frederick P Brooks Jr. 1995. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[22] M Cataldo, M Bass, J D Herbsleb, and L Bass. 2007. On Coordination Mechanisms in Global Software Development. In *International Conference on Global Software Engineering (ICGSE 2007)*. 71–80. DOI:http://dx.doi.org/10.1109/ICGSE.2007.33

[23] Maelick Claes, Tom Mens, Roberto Di Cosmo, and Jérôme Vouillon. 2015. A Historical Analysis of Debian Package Incompatibilities. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 212–223. http://dl.acm.org/citation.cfm?id=2820518.2820545

[24] Ivica Crnkovic. 2001. Component-based software engineering new challenges in software development. *Software Focus* 2, 4 (2001), 127–133. DOI:http://dx.doi.org/10.1002/swf.45

[25] Cleidson R B de Souza and David F Redmiles. 2008. An Empirical Study of Software Developers' Management of Dependencies and Changes. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 241–250. DOI:http://dx.doi.org/10.1145/1368088.1368122

[26] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

[27] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. 2015. 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 (ECSAW '15)*. ACM, New York, NY, USA, 41:1—-41:6. DOI: http://dx.doi.org/10.1145/2797433.2797476

[28] A Decan, T Mens, M Claes, and P Grosjean. 2016. When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 493–504. DOI:http://dx.doi.org/10.1109/SANER.2016.12

[29] Stephen G Eick, Todd L Graves, Alan F Karr, J S Marron, and Audris Mockus. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. Softw. Eng.* 27, 1 (jan 2001), 1–12. DOI:http://dx.doi.org/10.1109/32.895984

[30] A Gelbukh. 2011. *Computational Linguistics and Intelligent Text Processing: 12th International Conference, CICLing 2011, Tokyo, Japan, February 20-26, 2011. Proceedings*. Springer. https://books.google.com/books?id=I840zhFc

[31] D M German, B Adams, and A E Hassan. 2013. The Evolution of the R Software Ecosystem. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. 243–252. DOI:http://dx.doi.org/10.1109/CSMR.2013.33

[32] RE Grinter. 2001. From local to global coordination: lessons from software reuse. *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work* (2001), 144–153. DOI:http://dx.doi.org/10.1145/500286.500309

[33] Stefan Haefliger, Georg von Krogh, and Sebastian Spaeth. 2008. Code Reuse in Open Source Software. *Management Science* 54, 1 (2008), 180–193. DOI:http://dx.doi.org/10.1287/mnsc.1070.0748

[34] Frank E Harrell Jr. 2016. *rms: Regression Modeling Strategies*. https://cran.r-project.org/package=rms

[35] James D Herbsleb and Audris Mockus. 2003. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Trans. Softw. Eng.* 29, 6 (jun 2003), 481–494. DOI:http://dx.doi.org/10.1109/TSE.2003.1205177

[36] JL Hodges and Erich L Lehmann. 1983. HodgesLehmann Estimators. *Encyclopedia of statistical sciences* (1983).

[37] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 92–101. DOI:http://dx.doi.org/10.1145/2597073.2597074

[38] John C Knight and Michael F Dunn. 1998. Software quality through domain-;driven certification. *Annals of Software Engineering* 5, 1 (1998), 293–315. DOI:http://dx.doi.org/10.1023/A:1018960021044

[39] Alexandra Kuznetsova, Per Bruun Brockhoff, and Rune Haubo Bojesen Christensen. 2016. *lmerTest: Tests in Linear Mixed Effects Models*. https://cran.r-project.org/package=lmerTest

[40] Daniel Le Berre and Pascal Rapicault. 2009. Dependency Management for the Eclipse Ecosystem: Eclipse P2, Metadata and Resolution. In *Proceedings of the 1st International Workshop on Open Component Ecosystems (IWOCE '09)*. ACM, New York, NY, USA, 21–30. DOI:http://dx.doi.org/10.1145/1595800.1595805

[41] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 242–251. DOI:http://dx.doi.org/10.1145/2597073.2597109

[42] Mircea Lungu, Romain Robbes, and Michele Lanza. 2010. Recovering Inter-project Dependencies in Software Ecosystems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 309–312. DOI:http://dx.doi.org/10.1145/1858996.1859058

[43] Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering* 12, 5 (2007), 471–516. DOI:http://dx.doi.org/10.1007/s10664-007-9040-x

[44] Shinichi Nakagawa and Holger Schielzeth. 2013. A general and simple method for obtaining R2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution* 4, 2 (2013), 133–142. DOI:http://dx.doi.org/10.1111/j.2041-210x.2012.00261.x

[45] Jeroen Ooms. 2013. Possible Directions for Improving Dependency Versioning in {R}. *CoRR* abs/1303.2 (2013). http://arxiv.org/abs/1303.2140

[46] J Ossher, S Bajracharya, and C Lopes. 2010. Automated dependency resolution for open source software. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 130–140. DOI:http://dx.doi.org/10.1109/MSR.2010.5463346

[47] R Development Core Team. 2008. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.r-project.org

[48] Franklin E Satterthwaite. 1946. An approximate distribution of estimates of variance components. *Biometrics bulletin* 2, 6 (1946), 110–114.

[49] Holger Schielzeth and W Forstmeier. 2009. Conclusions beyond support: overconfident estimates in mixed models. *Behavioral Ecology* 20, 2 (2009), 416–420. DOI:http://dx.doi.org/10.1093/beheco/arn145

[50] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2013. A Study of Library Migration in Java Software. *CoRR* abs/1306.6 (2013). http://arxiv.org/abs/1306.6262

[51] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 356–366. DOI:http://dx.doi.org/10.1145/2568225.2568315

[52] Ivo van den Berk, Slinger Jansen, and Lútzen Luinenburg. 2010. Software Ecosystems: A Software Ecosystem Strategy Assessment Model. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (ECSA '10)*. ACM, New York, NY, USA, 127–134. DOI:http://dx.doi.org/10.1145/1842752.1842781

[53] Georg von Krogh, Sebastian Spaeth, and Karim R Lakhani. 2003. Community, joining, and specialization in open source software innovation: a case study. *Research Policy* 32, 7 (2003), 1217–1241. DOI:http://dx.doi.org/10.1016/S0048-7333(03)00050-7

[54] Wei Wang and Michael W Godfrey. 2013. Detecting API Usage Obstacles: A Study of iOS and Android Developer Questions. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 61–64. http://dl.acm.org/citation.cfm?id=2487085.2487100

[55] Frank Wilcoxon and Roberta A Wilcox. 1964. *Some rapid approximate statistical procedures*. Lederle Laboratories.

[56] Chris Williams. 2016. How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. (2016). http://www.theregister.co.uk/2016/03/23/npm

[57] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2015. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering* (2015), 1–47. DOI:http://dx.doi.org/10.1007/s10664-015-9411-7

[58] Yunwen Ye, Kumiyo Nakakoji, and Yasuhiro Yamamoto. 2007. Reducing the Cost of Communication and Coordination in Distributed Software Development. In *Proceedings of the 1st International Conference on Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD'07)*. Springer-Verlag, Berlin, Heidelberg, 152–169. http://dl.acm.org/citation.cfm?id=1778650.1778663

[59] Mahmood Shafeie Zargar. 2013. Reusing or Reinventing The Wheel: The Search-Transfer Issue in Open Source Communities. In *Proceedings of the International Conference on Information Systems, {ICIS} 2013, Milano, Italy, December 15-18, 2013*. http://aisel.aisnet.org/icis2013/proceedings/ResearchInProgress/101