"My Code is Broken But It's Not My Fault": Coordination Costs of Managing External Dependencies

Arun Kalyanasundaram, Christopher Bogart, Erik H. Trainer, James D. Herbsleb
Institute for Software Research
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
{arunkaly,cbogart,etrainer,jdh}@cs.cmu.edu

Abstract—**Modern day software development is about building new software products fast and with fewer lines of code. This is often accomplished by reusing existing code available in the form of libraries, packages and other formats, thereby creating dependencies across projects. However, developers often have no idea what long-term costs are associated with managing dependencies. In this paper, we ask how frequent and how costly are problems of dependency management? We study the R ecosystem on Github comprising of 3,812 R package repositories containing 69,491 issues. We find that a surprisingly large proportion (24%) of issues are problems caused by external dependencies or arise as a result of managing them. We call these issues external problems, and estimate the costs they incur in terms of the coordination required to resolve them. We operationalize coordination costs as the time taken to close, the number of comments and the number of people involved in an issue. We find that external problems take 34% longer and require 26% more discussions than other problems. We also show that certain policies and practices of projects reduce the likelihood of external problems. Our findings show that, not only are external problems more common but are also difficult to deal with.**

Keywords-Software reuse; Dependency management; R ecosystem; GitHub; External dependencies; Coordination Costs

I. Introduction

On March 22 2016, several thousand JavaScript developers and companies including Facebook could not build their code and worse, were unable to update their apps and services running on the web [1]. This happened because a developer decided to remove his module, *left-pad* from NPM (NodeJS Package Manager), which is a popular package manager for JavaScript projects. Thousands of projects were using this module by taking a dependency on it either directly or indirectly. Third-party dependencies such as these are commonly referred to as *upstream* or *external* dependencies. This example, although an extreme case, shows the impact that external dependencies can have on software projects. It illustrates how fragile and complex the web of dependencies is in modern day software development. Moreover, social coding platforms like GitHub [2] are making it easier for developers to reuse existing code, thereby encouraging projects to take on more dependencies.

While reusing code (by taking on dependencies) helps developers quickly integrate new functionalities into their projects [3, 4], it is not without its challenges, such as package conflicts [5, 6], API incompatibilities [7], versioning [8], [9] and library migration [10]. There are commercially available tools to help with some of these challenges, such as Greenkeeper [11] and Gemnasium [12] which help developers manage dependencies by notifying changes in upstream dependencies.

Despite the breath of study of dependency management in the software engineering literature [13]–[17], we know very little about the overall impact external dependencies can have on a software project. Therefore, we take a step back and ask how frequent and how costly are problems of dependency management? This is not only useful to evaluate dependency management solutions but also for effectively quantifying tradeoffs related to software reuse [18, 19] and to assess long-term consequences of external dependencies.

The goals of this paper are threefold: (1) to estimate the extent to which problems due to external dependencies occur, (2) to identify and evaluate various indicators of these problems, and (3) to measure the costs incurred in resolving these problems.

Our approach involves the following three phases. First, we curate a dataset of 69,491 issues (including pull requests) from 3,812 repositories on GitHub and implement a keyword based filtering technique [10, 20] to identify issues that are caused by external dependencies. We call these issues *external problems*, and found that about 24% of all issues were external problems. Second, we use a qualitative content analysis approach [21] to identify indicators (e.g. certain characteristics of dependencies, project practices, etc.) of external problems. We fit a zero-inflated negative binomial regression model [22, 23] to evaluate the relationships between our identified predictors and the number of external problems in a project.

In the final phase, we estimate the costs incurred by external problems in terms of the coordination required to resolve them. We operationalize coordination costs as the time taken to close an issue and the number of discussions (comments) involved. We fit a linear mixed-effects model [24] to determine whether coordination costs of external and internal problems (remaining issues not classified as external) differ. We found that external problems take about 34% longer and have 26% more comments. This, to the best of our knowledge is the first study that gives a meaningful estimate of the costs associated with managing external dependencies relative to the cost of internal maintenance. This, along with developing a generic approach to assess the impact of external dependencies and evaluating the relationships between various predictors and the number of external problems in a project are the key research contributions of our paper.

The rest of the paper is organized as follows. In Sec. II we discuss related work and introduce our research questions. We briefly describe our methodology in Sec. III. We present our results along with a detailed explanation of our models in Sec. IV. We discuss the practical implications of our work, threats to validity and steps to replicate in Sec. 0. We conclude with scope for future work in Sec. VI.

II. Background And Related Work

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 [25], packages [26] and other formats, which are broadly referred to as *upstream* (or *external*) dependencies. The benefits of software reuse [27] are quite evident as it helps developers quickly integrate new functionality [3, 4, 28] even with limited skills [29], and improves reliability [30] since code used and tested by many people may have fewer defects.

However, there are a number of challenges with managing external dependencies [13, 15, 31]. Of these challenges, the most widely studied is change management, the process by which changes made to *upstream* dependencies affect *downstream* projects. We know that change in large successful software projects is inevitable [32]. However, the impact on downstream projects due to upstream changes can include severe problems such as failed installation due to version conflicts [8], failed build due to API incompatibilities [7], newly introduced bugs that otherwise would not exist [33] and impacts on performance [10].

Naturally, several practices and tools have been employed to address these problems. They can be broadly classified into three types: (1) *Ecosystem policies*: Software projects are often part of an existing ecosystem, a set of actors interacting with a shared market for software and services [34]. The ecosystem enforces policies or encourages certain best practices for managing upstream dependencies. For example, the CRAN (Comprehensive R Archive Network) ecosystem [20] for R packages requires that upstream developers communicate with developers of downstream projects before making API breaking changes [35]. (2) *Team / Developer practices*: Communication between upstream and downstream developers is an important aspect of effective change management. This can happen either by upstream developers notifying downstream projects or vice versa [15]. (3) *Tools*: Several tools such as Greenkeeper [11] and Gemnasium [12] have been developed to notify downstream projects of upstream changes.

However, apart from change management, downstream projects have to deal with a myriad of other problems associated with upstream dependencies. For example, if a particular library becomes obsolete, finding and migrating to a new library could become an overwhelming task [10]. Another example is downstream projects facing difficulties using a library because of insufficient API documentation [36]. Although many solutions exist to address these problems, we do not know the extent and costs of resolving these problems, which is required to evaluate dependency management solutions.

Previous studies have estimated the extent of specific types of dependency problems. Claes et al. [5] studied the extent of Debian package conflicts and found that the fraction of strong conflicts remains constant over time. Wu et al. [7] found that in Apache and Eclipse ecosystems, missing classes and methods happened more frequently than missing interfaces with new framework releases. Decan et al. [37] found that about 41% of build errors on CRAN were due to dependency updates. However, there is no single approach to measure the extent and costs of all types of problems associated with dependency management. Since social coding platforms such as GitHub are making it easier for developers to search and integrate existing code [38], our questions are more relevant now than ever before.

For brevity, we formally define a *problem* in a software project as a scenario that requires one or more developers (or users) work towards arriving at a solution. Therefore, we call *problems* that are caused by upstream dependencies or arise as a result of managing them as *external problems,* and all other problems as *internal problems*. This terminology helps us communicate our research questions in a more general yet precise way. We ask:

RQ1: How prevalent are external problems?

Knowing the extent of external problems does not tell us about the reasons behind their occurrence. Therefore, we ask:

RQ1A: What are the different kinds of external problems and why do they occur?

Since the number of external problems may vary from one project to another, it is important to identify and evaluate key predictors of external problems. Therefore, we ask:

RQ1B: What are the predictors of the number of external problems in a project?

Although previous studies acknowledge that downstream projects incur "upgrade costs" due to changes in upstream dependencies [7] and that managing dependencies requires considerable "extra work" [39], these are broadly referred to as effort, and no operationalization of cost is provided. Since resolving a problem often requires coordination among various stake-holders [40], and since communication and coordination costs are a considerable proportion of costs in software development [41]–[43], we ask:

RQ2: What are the coordination costs of external problems?

Coordination costs manifest in the form of delay (time taken to resolve an issue) [44], number of people involved [44, 45] and number of discussions (or comments) in an issue [41, 46]. Since we are studying coordination costs in open-source software projects, it seems appropriate to focus on indicators of effort, rather than financial costs. In order to make these measures of effort more meaningful, we express it in terms of relative effort, by comparing the coordination costs of external and internal problems, and evaluating if the difference is significant. Therefore, we ask:

RQ2A: Do external problems incur higher coordination costs than internal problems?

III. Methodology

Although our research questions are not specific to any particular ecosystem, we choose to use data from the R ecosystem [47]. R is a well-established ecosystem with thousands of packages that are used in several domains [20, 26]. An R package is a standardized format to easily distribute compiled code [48], which other packages (and scripts) can create a dependency on in order to use the functionality it offers. While choosing one particular ecosystem limits generalizability, it allows us to provide a richer account, and reduces the risk of potential confounds in our findings [25, 49].

A. Dataset

While R package binaries are primarily distributed through CRAN and BioConductor, they are developed on R-Forge and GitHub. However, there is evidence that the popularity of R-Forge is on the decline and GitHub is emerging as a major platform for the development of R-packages [26]. In addition, GitHub is also used as a distribution channel for R packages that are not in CRAN or BioConductor [26]. Therefore, we chose to use data from Github, and mined all R related public repositories (excluding forks) that were created before February 2016. We used GitHub Linguist [50] to identify R related repositories, and found 117,586 repositories. Many of them, however, contain R code written for class projects / homework, scripts written for personal use, etc. We therefore, restricted our sample to repositories that were only developing R packages, resulting in 19,233 repositories. Previous research shows, however, that many R package repositories on GitHub are just mirrors [26] of their corresponding binaries distributed on CRAN and Bioconductor. Filtering out mirrored repositories gave us 10,550 repositories.

B. Identifying External Problems

GitHub's issue tracker is a tool primarily used to report bugs and submit pull requests [51, 52]. It is sometimes used to ask questions [39] and propose new ideas [53]. For convenience, we call entries in the issue tracker that are not pull requests as bug reports, and we refer to both pull requests and bug reports as *issues*. Earlier we introduced the notion of *problems* in a project as a generic term to refer to scenarios that require one or more developers (or users) work towards arriving at a solution. Therefore, in order to answer our research questions, we consider the *issues* in a repository as *problems* in a project, an approach similar to Blincoe et al. [45] where bug reports were treated as tasks. In addition, we treat each repository as a project, because they are all individual R packages that are self-contained and can often be distributed and consumed independently. Therefore, throughout this paper, the terms *repository* or *project* mean the same thing. We also use the terms *issues* and *problems* interchangeably. For brevity, we often just use the term *dependencies* to refer to upstream or external dependencies.

Among the 10,550 repositories in our dataset, we found about 3,812 repositories with at least one issue. These repositories contained 69,491 issues (51,054 bug reports and 18,437 PRs) in total that were created before February 2016. The key preliminary step in our analysis is to classify a given problem as external or internal. We use a keyword based filtering technique previously used to identify whether a given mailing list discussion was related to a set of R packages [20]. Our approach is based on the premise that if the name of at least one upstream dependency is never mentioned in the discussions of a problem then the problem is *not* an external problem. Identifying the set of upstream dependencies in an R package is straight forward, either using the DESCRIPTION file or looking at import statements in source code [20]. If the content (title, body and comments) of a given issue in our dataset contains the name of an upstream dependency, then it is classified as an external issue.

A false positive occurs when our approach incorrectly classifies an issue as external. Our approach could result in false positives when the name of an upstream dependency happens to be an English dictionary word. However, we found very few names were dictionary words and their usage in the context of a regular English sentence in an issue discussion is uncommon. On the other hand, we found that a major source of false positives is the use of dependency names inside code snippets posted in issue discussions. Fortunately, GitHub provides a standardized markdown [54] that users use to annotate code snippets in issue discussions. Making use of this, our classification approach filters out code snippets from the contents of an issue, and then classifies an issue as external if the name of at least one upstream dependency of the project appears in its contents. We evaluated our approach on a random sample of 500 issues (390 bug reports and 110 PRs). Our approach classified 127 as external, however, a manual inspection resulted in 24 false positives and 14 false negatives. This gives us a precision of about 0.81, recall of 0.88, and a reasonably good F-score of about 0.84 (based on other text classification studies [55]).

C. Mixed-Methods Approach

We use a mixed-methods approach with a sequential exploratory strategy [56] for the following two reasons. First, our research questions require understanding the different kinds of external problems and the reasons why they occur, for which a qualitative analysis is most appropriate [57, 58]. Second, a qualitative analysis helps us generate hypotheses that we later test using a quantitative analysis.

We fit a zero-inflated negative binomial regression [22, 23] model to evaluate the relationships between various predictors (identified using our qualitative analysis) and the number of external problems in a project. We then fit a linear mixed-effects model [24] to estimate the differences in coordination costs between external and internal problems. In order to make it easier for the reader, we provide the details of our methods in conjunction with their corresponding results in Sec. IV.

IV. Method Details And Results

A. RQ1: How Prevalent are External Problems?

Based on our approach to classify *external* issues, we found that out of 69,491 issues in our dataset, 16382 issues (or about 24%) were categorized as external. TABLE I. shows the proportion of external problems (with actual numbers in parenthesis) among bug reports vs. pull-requests and with state open vs. closed.

TABLE I. Proportion of External Problems

	Closed / Merged	Open	Total
Bug Reports	28% (10,265)	22% (3,142)	26% (13,407)
Pull Requests	16% (2,814)	20% (161)	16% (2,975)
Total	24% (13,709)	22% (3,303)	24% (16,382)

There are two key takeaways from TABLE I. First, it shows that problems due to external dependencies are a substantial proportion of problems reported on the issue tracker. Second, the higher percentage of external problems among bug reports (26.3%) as opposed to pull requests (16%) shows the potency of external problems, since bug reports are generally considered as undesirable costs whereas pull requests are often used as a standard mechanism for code contribution.

These results show that a substantial proportion of issues are external problems, hence it is worthwhile to identify the different kinds of external problems and the reasons why they occur.

B. RQ1A: Different Kinds of External Problems

In order to identify and catalog the different kinds of external problems, we performed a qualitative content analysis [21, 59] on a random sample of 105 external issues (82 bug reports and 23 PRs). We started with a subset of 35 issues, and through a process called open coding [60] we assigned codes to issue discussions to elicit the type of problem being discussed. We then linked these codes to emerging categories, a process called axial coding. We repeated the whole process, iterating every time with another subset of issues until no new categories emerged. We use our qualitative analysis to generate hypotheses and identify control variables that should be included in our quantitative analysis. Broadly, we found four different kinds of external problems as discussed below.

1) Version Management: Typically, when a project incorporates an external dependency, it uses the latest version. Over time, however, new versions of a dependency might become available. These new versions often contain bug fixes, performance optimizations or new features and so, to use these changes a project has to constantly update itself to use newer version of its dependencies.

In our qualitative content analysis, we found that developers proactively update to newer versions of dependencies by creating issues to track the required changes, sometimes even before the new version is released. We also found a few instances where developers of upstream dependencies informed downstream developers of changes before they released a new version.

Interestingly, R allows end users to use an earlier version of dependencies of a package even though the package supports the latest version of dependencies [8]. This is especially common in R because certain packages are used for results in scientific publications and since updating the version of packages could change these results, end users continue to use earlier versions. Therefore, projects have to make sure they also support earlier versions of dependencies. In our content analysis, however, we found that as a project evolves, it could become incompatible with earlier versions of its dependencies. So, on one hand the project has to cater to the needs of users who use earlier versions, whereas on the other hand it requires considerable effort to support earlier versions.

To summarize, there are primarily two reasons why this type of external problem occurs: a) the need to support latest versions of dependencies, and b) incompatibilities with specific versions of dependencies. In both cases, each new version is an opportunity for new incompatibilities to arise and for problems to occur as the incompatibilities are overlooked or adaptations are performed incorrectly. Therefore, we hypothesize:

H1: The number of external problems in a project increases with the increase in average number of releases (per year) of its dependencies.

It is also intuitive that projects that have more dependencies will tend to have more external problems:

H2a: The number of external problems in a project increases with the increase in its number of dependencies.

Projects may adopt certain policies and practices that reduce the likelihood of external problems. For example, R packages hosted on CRAN are curated, and must communicate API breaking changes to all downstream packages (that are also on CRAN) before making a release [35]. Therefore, one could hypothesize that because of a set of policies and practices, taking a dependency on a CRAN package would lead to fewer external problems than other upstream dependencies not in CRAN. For brevity we call them CRAN and nonCRAN dependencies respectively:

H2b: The number of external problems in a project increases with the increase in its number of CRAN dependencies but at a lower rate compared to its nonCRAN dependencies.

Since each project in our dataset is also an R package, we can make a similar argument, that since they receive the benefit of change management practices they should experience fewer external problems:

H3: Projects that are CRAN Packages have fewer external problems than other projects.

2) Installing Dependencies: When a user installs a software product (say an R package), she is also required to install its dependencies. Although in R this process is automated, it is still possible to encounter errors [37]. In our qualitative analysis, we found that users reported installation errors on the software project's issue tracker on GitHub even when the errors indicated that there were problems installing one or more of its dependencies. We found three primary reasons for this type of external problem to occur: a) conflicts between function names across different packages, b) in-compatibility with operating systems (and the version of R), and c) install instructions are inaccurate or outdated.

Since packages could be made more robust over time, this type of problem could occur less frequently with dependencies that have been around for a long time. Therefore we control for two different measures of maturity of a project's dependencies: average age and average major version number.

3) Defects or Limitations in Upstream: Surprisingly, our content analysis revealed that a number of external problems in a project were defects in the upstream dependencies. However, when an external problem was reported in a project, we found that the onus was on the project developers to investigate and take necessary actions. In most cases, the developers asked the reporter to open a defect in the upstream dependency's issue tracker (or other available channels). Moreover, in certain cases the project developers also made code changes to upstream dependencies to resolve such problems. In any case, these projects incur significant coordination costs as a result.

It is widely known that projects with more developers and end users get more defects reported [61, 62], especially when they use the software in different ways and in different contexts. We do not know, however, if this is also applicable for external problems. Since each project in our dataset is an R package, a reasonable measure of the number of different kinds of use is the number of repositories that have taken a dependency on it. These are called *downstream repositories*. Therefore, we hypothesize:

H4: The number of external problems in a project increases with the increase in number of its downstream repositories.

Previous research has shown that one of the reasons why developers multitask across projects is to contribute code to upstream dependencies [63]. However, we do not know if more external problems in a project drives its developers to contribute more to upstream projects. Therefore, we hypothesize:

H5: Projects with more external problems have more developers contributing to upstream projects.

4) Adding / Removing Dependencies: We found in our qualitative analysis that it is quite common for developers to discuss with others before adding a new dependency. This was accomplished by creating issues to discuss about the pros and cons of choosing a particular dependency.

Existing dependencies, however, were sometimes removed and/or replaced. In our sample, we found that a developer removed a dependency because she only required a small part of the functionality offered by the dependency, which she felt was easier to maintain a copy in her own project. On the other hand, we found developers replaced certain dependencies because they had other better performing alternatives that offered similar functionalities. Therefore, these are another form of external problems since they are triggered by one or more upstream dependencies, and as a result, incur significant coordination costs.

The process of adding, removing and replacing dependencies is one type of churn in a project. Prior research has shown that churn is a strong predictor of number of defects in a project [64]. Since R packages modify the DESCRIPTION file [20] whenever dependencies are added, removed or changed, we use the number commits made to the DESCRIPTION file per year as our measure of churn, which we use as a control variable.

C. *RQ1B: Predictors of External Problems*

In this section, we build a regression model to test our hypotheses we generated in the previous section and to study the relationships between our identified predictors and the number of external problems in a project.

1) Zero-Inflated Model: Our unit of analysis is a repository and from among the 3,812 repositories in our dataset, we removed 651 repositories that had missing values for at least one covariate, giving us 3,161 repositories to analyze. We did not find any significant differences in the mean age and size of filtered and remaining repositories. Along with the relatively small proportion of repositories with missing data, this gives us confidence our sample is not biased. TABLE II. gives the descriptive statistics of all the variables in our model before any log transformations.

TABLE II. DESCRIPTIVES OF VARIABLES BEFORE TRANSFORMATION

Statistic	**Mean (St.Dev)**	**Median (Min,Max)**
#External Problems (DV)	5.04 (23.94)	1 (0, 533)
Avg. Upstream Releases / year	2.92 (1.16)	2.68 (0.13, 14)
#Total Dependencies	8.45 (6.85)	7 (1, 105)
#Cran Dependencies	6.83 (5.59)	5 (1, 102)
#nonCran Dependencies	1.62 (2.75)	1 (0, 32)
Avg. Upstream Age in years	5.71 (2.12)	5.68 (0.9, 15)
Avg. Upstream Major Version Number (plus one)	2.06 (1.19)	2 (1, 50)
DESCRIPTION File Commits per year	14.37 (33.3)	6.54 (0, 1273.33)
#Internal Problems	15.57 (51.6)	4 (0, 1212)
#Downstream repositories	47.76 (627.28)	1 (0, 21481)
#Developers	2.26 (2.60)	2 (1, 30)
Fraction of Developers Contributing to at least One Upstream Dependency	0.12 (0.23)	0 (0, 1)
#Stars	14.9 (73.7)	2 (0, 1733)
#Lines of Code	1712.8 (14997)	577 (0, 828903)
Repository Age in Years	2.06 (1.22)	1.76 (0.26, 8.3)
N = 3,161		

In order to control for project age, size and popularity, we use the age, number of lines of code, and number of stars of a repository respectively [46]. We also wanted to include the number forks and watchers but found them to be highly correlated with the number of stars, which is in line with previous findings [46]. We decided not to include the number of source files and code comments as they were highly correlated with the number of lines of code. We also use the number of developers in a repository as an additional measure of size [46]. We found a value less than 0.5 for all pairwise correlations among our continuous variables, which is less than the customary threshold of 0.7 [65], which would trigger further investigation of multicollinearity.

Since our outcome variable is based on a count, an OLS regression (even with a log transformation) may not be appropriate. In addition, we found evidence of over dispersion in our data, which indicates the use of a negative binomial regression [23]. However, our outcome variable has a large proportion of zeros (~45%), and therefore, we evaluate the suitability of a zero-inflated model [22]. A large fraction of zeros alone does not warrant using a zero-inflated model. There has to be a plausible independent process behind the occurrence of zero values. For example, a project may only use standard and popularly known dependencies, and therefore, no external problems get reported. The goal of a zero inflated model is to combine a count model with a logit model that estimates the likelihood of an excess zero (a zero caused by another process that cannot be explained by the count model).

Moreover, we found using Vuong's non-nested test [66] that the zero inflated model is a better fit for our dataset. The count process is modeled using a negative binomial with log link, whereas the likelihood of an excess zero is modeled using a logit model. We use the zero-inflated model implementation in the *pscl* package (version 1.4.9) [67] in R. We also log transformed (natural logarithm) a few covariates (adding 0.5 to account for zero value) as needed to improve the model fit. We found the VIF (variance inflation factors of our estimated coefficients to be below three, which is below the recommended maximum value of five [68], indicating the absence of multi-collinearity.

We use the standard control variables: size, age and popularity of a repository [46]. We use the number of developers involved and amount of code written as measures of size, and the number of stars as a measure of popularity.

2) *Model Results:* TABLE III. shows the results of our zero-inflated model where the outcome variable is the number of external problems in a project (repository). The model has two parts, a) the count part shows the negative binomial regression coefficients for each variable and their standard errors (in parenthesis), and b) the zero-inflation part shows the logit coefficients for each variable, which are the log odds for predicting excess zeros. Therefore, a positive or a negative zero-inflation coefficient tells us whether the likelihood of an excess zero increases or decreases respectively with increase in the value of the corresponding variable. These two parts can be interpreted independently. For brevity, we use abbreviated variable names in TABLE III.

TABLE III. ZERO INFLATED MODEL. THE RESPONSE IS THE NUMBER OF EXTERNAL PROBLEMS IN A PROJECT.

	Count	Zero-Inflation
(Intercept)	-1.681*** (0.180)	2.796* (1.181)
Deps. Releases (H1)	0.072*** (0.024)	0.012 (0.137)
#Cran Deps. (H2b)	0.024*** (0.004)	-0.840*** (0.181)
#nonCran (H2b)	0.038*** (0.008)	-0.537* (0.239)
IsCRAN (H3)	-0.024 (0.054)	0.709 (0.443)
Deps. Age	-0.023 (0.014)	-0.164 (0.102)
Deps. Version	-0.011 (0.020)	-0.007 (0.068)
DESC. Commits (log)	0.147*** (0.030)	-0.567* (0.270)
#Internal Probs. (log)	0.554*** (0.021)	0.471** (0.166)
#Downstream (log) (H4)	0.055*** (0.018)	0.111 (0.157)
#Developers (log)	0.170*** (0.051)	-1.121* (0.544)
Upstream Contrib. (log) (H5)	0.423*** (0.093)	0.164 (0.464)
#Stars (log)	0.232*** (0.018)	-0.432** (0.150)
LOC (log)	0.052* (0.023)	0.393* (0.165)
Repo Age (log)	0.069 (0.070)	-2.494** (0.791)
Log(theta)	0.378*** (0.057)	

Log Likelihood: -5415 on 31 Df
AIC: 10891.19
Num. of Observations: 3161
Number of iterations in BFGS optimization: 59
$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

In the count part, the coefficients are log linked and therefore, we interpret them by taking their exponential. For example, one of the characteristics of upstream dependencies we chose was the average number of releases it made per year (H1). This predictor has a coefficient of 0.072 as shown in TABLE III. Therefore, a one unit increase is associated with 1.075 ($e^{(0.072)}$) times increase in the expected number of external problems, which in other words is a 7.5% increase on average in the number of external problems, holding other variables constant.

We note that the model presented in TABLE III. does not include the covariate #TotalDependencies. Since #TotalDependencies is a sum of #CRAN and #nonCRAN dependencies, it is collinear with these two covariates. Therefore, including it in the same model would make the coefficients difficult to interpret. Hence, we fitted a second zero-inflated model which contained #TotalDependencies but excluded the covariates #CRAN and #nonCRAN dependencies. The #TotalDependencies in this model had a coefficient of 0.027*** (std. err: 0.004), and the coefficients of other variables more or less remained the same as the values in TABLE III.

We use our model to test the hypotheses we generated in the previous section. The results and their corresponding interpretations are summarized in TABLE IV.

TABLE IV. SUMMARY OF HYPOTHESES TESTS AND INTERPRETATION OF COEFFICIENTS

Hypothesis	Supported? (at p < 0.05)	Interpretation (*Holding other variables constant, the change in the expected number of external problems in a project…*)
H1 (External problems increase with #upstream releases)	Yes	for a one unit increase in the average number of releases of upstream dependencies per year is a 7.5% ($e^{(0.072)}$) increase.
H2a (External problems increase with #total dependencies)	Yes	for a one unit increase in the number of dependencies is a 2.74% ($e^{(0.027)}$) increase. (Coefficient not shown in TABLE III.)
H2b (CRAN deps. have a lower positive coefficient than nonCRAN deps.)	Yes	for a one unit increase in CRAN deps. is a 2.4% ($e^{(0.024)}$) increase, whereas for a one unit increase in nonCRAN deps. is a 3.87% ($e^{(0.038)}$) increase. The difference in the coefficients is significant because there is less than fifty percent overlap in their 95% bootstrap confidence intervals.
H3 (CRAN package projects have fewer external problems)	Not Statistically Significant	for a CRAN package project is a 2.4% ($1-e^{(-0.024)}$) decrease compared to other projects.
H4 (External problems increase with no. of downstream repositories)	Yes	for a 1% increase (since the covariate is log transformed) in the average number of downstream repositories is a 0.06% increase.
H5 (Projects with more external problems have more developers contributing to upstrea)	Yes	for a 1% increase in the (average) fraction of developers contributing to upstream projects is a 0.42% increase.

Although it is evident that the total number of dependencies (H2a) should have a positive coefficient (0.027, not shown in TABLE III.), the difference between the coefficients of CRAN and nonCRAN dependencies (H2b) is not straightforward. Our model shows that the point estimate for the number of CRAN dependencies (0.024) is lower than the point estimate for the number of nonCRAN dependencies (0.038). Although both coefficients individually are statistically significant, in order to conclusively establish that the difference is also statistically significant we computed their bootstrap confidence intervals (with 1000 re-samples) [69]. The bootstrap technique is used to obtain a more robust nonparametric estimate of confidence intervals compared to the one computed from standard errors. We used unstandardized coefficients since the two variables are on the same scale. We found less than fifty percent overlap between their 95% confidence intervals: [0.015,0.032] and [0.022,0.056] for CRAN and nonCRAN dependencies respectively, thus establishing that the difference is statistically significant (at $p < 0.05$). Similarly, for projects that are CRAN packages (H3), the number of external problems is lower (-0.024) than other projects. However, this coefficient is not statistically significant, hence does not support our hypothesis H3.

Surprisingly, the fraction of developers contributing code to upstream dependencies has a strong positive association (0.443 on the log scale) on the number of external problems. Previous research shows that one of the main reasons developers work on multiple projects is because of inter-dependencies [63]. Our result supplements this finding since we show that it's not just the dependencies but the number of external problems as an important driver for developers to contribute code to upstream dependencies.

Finally, we interpret the zero-inflation part to provide possible explanations for excess zeros in the outcome variable. We see that the coefficients for both CRAN and nonCRAN dependencies are negative (-0.84 and -0.537) and statistically significant, which indicates that more the number of dependencies, the less likely an independent process (other than the count model) caused the project to have zero external problems. A possible explanation could be that when projects have very few dependencies, they are likely to be standard, widely used, and rarely cause problems.

To summarize, the key results of our model are: (1) Projects that depend on packages that make fewer releases are less likely to have external problems, (2) Taking dependency on a CRAN package leads to fewer external problems on average than taking dependency on a nonCRAN package, (3) Projects with more external problems have more developers contributing to upstream dependencies, (4) Projects with more downstream repositories are more likely to have external problems, and (5) Projects with fewer dependencies are more likely to not have any external problems at all.

The results of our model show the relationships of various predictors such as the characteristics of dependencies, project practices, and ecosystem policies with the number of external problems in a project. To the best of our knowledge this is the first study to quantitatively show this relationship.

D. *RQ2: Coordination Costs of External Problems*

We showed that a significant proportion (24%) of issues are external problems. However, in order to assess the impact of external problems, one must also measure the coordination costs associated with resolving them. Since our unit of an external problem is an issue, relevant measures of coordination costs are, (1) Time taken to resolve (close or merge) an issue [44], (2) the number of comments in an issue [41, 46], and (3) the number of people (unique users) involved in the discussions of an issue [44, 45].

Since by definition, our measures of coordination costs only apply to resolved issues, we retained 54,358 resolved issues (36,719 closed bug reports and 17,639 closed / merged pull-requests) from our initial dataset of 69,491 issues (51,054 bug reports and 18,437 PRs). In addition, we removed 549 issues where the creation and closing time were exactly same because they were issues migrated from other platforms such as *Google Code*. Therefore, this gives us 53,809 resolved issues from 3080 repositories. It is unlikely that this filtering process introduces any bias because our retained dataset is reasonably large and is about eighty percent the size of the original dataset. Among the set of 53,809 resolved issues, we found 13,043 external problems using our approach in Sec. III.B. We present the descriptive statistics of the three measures of coordination costs for these external problems in TABLE V.

TABLE V. DESCRIPTIVES OF COORDINATION COSTS OF EXTERNAL PROBLEMS

Variable	Mean (St.Dev)	Median (Min,Max)
Time Taken to Close/ Merge in days	49.07 (123.31)	3.81 (1.16e-5, 1.81e3)
No. of Comments	3.88 (5.98)	2 (0, 313)
No. of Unique Users	1.97 (1.07)	2 (1, 37)
N = 13,043		

TABLE V. shows that the time taken to close is highly dispersed with a mean of 49 days and a standard deviation of 123, which is more than twice the mean. In addition, the significantly low median of 3.81 days compared to the mean is indicative that most issues only take a few days to close, but there are several outliers that take extremely long to close. The external problems on average have about 3.88 comments (not including the body of the issue) and about 2 users (including the reporter) are involved in the discussions. In addition, at least half of the external problems have more than two comments and two unique users, indicating that there is a reasonable amount of back and forth discussions and coordination to resolve these problems.

However, certain types of external problems could incur higher or lower coordination costs. For example, we identified 4,385 issues in large projects (those with more than 10 developers), and found the mean time to resolve (63.26) and the number of comments (4.84) to be significantly higher (using a 1-sample t-test, $p < 0.001$) than their corresponding means given in TABLE V. Therefore, this illustrates the need to control for various confounding variables when comparing the coordination costs of external and internal problems.

E. *RQ2: Do External Problems Cost More than Internal?*

In order to make the measures of coordination costs more meaningful, we express it in terms of relative costs by comparing the coordination costs of external and internal problems. TABLE VI. shows that both the mean and median of all three coordination cost measures of external problems are higher than internal problems.

TABLE VI. COMPARISON OF COORDINATION COST MEASURES.

Variable	Mean		Median	
	External	*Internal*	*External*	*Internal*
Time	49.07	37.97	3.81	1.04
Comments	3.88	1.45	2	1
Users	1.97	1.47	2	1
N (External):13,043; N(Internal): 40,766				

If we can quantitatively establish that resolving external problems do in fact incur higher coordination costs than internal problems, it could significantly impact the way software developers use and manage dependencies.

1) Linear Mixed-Effects Model: Our goal is to estimate the difference in the measures of coordination costs between external and internal issues. Our unit of analysis is an issue, however, multiple issues can be part of a single repository. We have 53,809 issues belonging to 3080 repositories in our dataset. Therefore, to account for the variability across repositories, we fit a linear mixed effects model with a random intercept term for repository (uniquely identified with its GitHub URL). We also found that users report multiple issues, and our data has 6756 unique users reporting 53,809 issues. In order to account for the variability across issue reporters we include a random intercept term for reporter (uniquely identified by their GitHub user name). Our predictor is a dichotomous variable, isExternal, which is true when an issue is an external problem and false otherwise. Researchers in other fields [70, 71] have shown that not accounting for the variability in the coefficient of predictors across different groups could result in Type I errors (i.e. incorrectly claiming significant results) in mixed models. Therefore, we also include random slope terms for the predictor to vary across both groups - repositories and reporters.

The fixed effect factors in our model are at three levels - issue, repository and reporter. At the issue level we have, (a) *Is pull request* (true / false), (b) *Is enhancement* (true / false): if the label names on the issue contained either the word “enhancement” or “feature”. We decided not to filter out enhancements because we found in our qualitative analysis that certain types of external problems such as adding dependencies, upgrading versions were tagged as external problems. This implies that enhancements are not necessarily new features but could also be for issues that are more intricate. (c) *Number of cross-references*: An issue can contain one or more cross-references to other issues specified using GitHub’s Autolinked references (part of Flavored Markdown) [72]. We know that a cross reference between issues on two different repositories indicates that the two repositories are inter-dependent [73]. Similarly, cross-references among issues of the same repository could imply that the issues are inter-dependent, which could in turn increase their time taken to resolve. Therefore, we include the sum of all cross-references an issue has made or other issues have made to it as a control variable.

At the repository level we use widely used control measures [46] such as age, size (we use number of developers, lines of code) and popularity (number of stars) of the repository. In Sec. IV.C we showed that projects that took dependencies on CRAN packages were less likely to have external problems. Therefore, we also include whether the repository is a CRAN package or not as a control variable. At the reporter level, we control for whether the reporter is a contributor: made at least one commit to the repository of the reported issue. This follows from previous research that shows developer submitted pull requests are more likely to be accepted [46].

Previous research has shown that delay in software development activities increases with the number of people involved [44]. Similarly, one could also argue that the issues that take longer and have more comments [46] are a result of more users involved in it. Therefore, we only use the number of comments and time taken to resolve an issue as our measures of coordination costs (outcome variables), and we use the number of unique users involved in an issue as a control variable.

Our two linear mixed-effect models (one for each outcome measure) are implemented using the lmerTest [74] package in R, which is built on top of the lme4 package [24] but provides p-values based on Satterthwaite's approximations [75]. The effect size of coefficients is computed using the ANOVA analysis. We found no evidence for multi-collinearity as inferred from the VIF implementation [76] for mixed models adapted from the *rms* package [77] in R, since VIF of our estimated coefficients were below 2.5. We compute the marginal (R^2_m) and conditional (R^2_c) coefficient of determination as implemented in the *MuMIn* package [78, 79] 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. In addition, we log transformed all our continuous independent and dependent variables to ensure normality and account for the high dispersion.

2) Model Results: TABLE VII. shows the fixed effect coefficients for both models (time and comments model). The fixed effect coefficient of the predictor "*Is External*" is statistically significant ($p < 0.001$) in both models and shows that on average external problems take about 34% ($e^{.294} - 1$) longer to close and have 26% ($e^{.234} - 1$) more comments, holding other variables constant. TABLE VII. also shows the sum of squares for each variable and we note that the "Is Pull Request" variable in the time model has a value of 68033, which accounts for almost 70% of the overall variance explained. It also has a large negative coefficient (-3.46), which indicates that pull requests on average take about 97% less time to close (or merge) compared to bug reports.

TABLE VII. FIXED EFFECT COEFFICIENTS OF OUR TWO LINEAR MIXED EFFECTS MODELS WITH OUTCOME VARIABLES LOG TRANSFORMED

	Outcome: Log (Time to close)		**Outcome: Log (#Comments)**	
	Coeff. (Std. err)	***Sum Sq.***	***Coeff. (Std. err)***	***Sum Sq.***
(Intercept)	-3.156*** (0.190)		-1.118*** (0.027)	
Is External	0.294*** (0.048)	341	0.234*** (0.009)	163.1
#Unique Users (log)	2.204*** (0.055)	14697	2.184*** (0.010)	14655.9
Is Pull Request	-3.464*** (0.041)	68033	-0.301*** (0.007)	558
Is Enhancement	1.603*** (0.051)	9103	0.009*** (0.008)	29.7
#Cross Refs. (log)	0.463*** (0.031)	1974	0.201*** (0.005)	379.4
Is Reporter a Developer	0.914*** (0.052)	2799	0.075*** (0.009)	20.6
Repo: age in years	0.899*** (0.092)	862	0.015 (0.012)	0.4
Repo: #Stars	0.056* (0.028)	36	0.018*** (0.004)	5.6
Repo: #Developers	0.078 (0.070)	11	-0.023* (0.010)	1.6
Repo: LOC	0.103*** (0.027)	129	0.007* (0.003)	1.1
Repo: is CRAN	0.048 (0.081)	3	0.016 (0.011)	0.6
AIC / BIC / Log. Lik	276935.7 / 277104.7 / -138448.9		86560.9 / 86729.9 / -43261.5	
R^2_m / R^2_c	0.27 / 0.46		0.66 / 0.72	
Num. of Observations: 53809 $^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$				

The results show that both models fit the data well: (1) The R^2_m value for the *time* model in TABLE VII. shows that fixed effects explain about 27% of the variability in the data, and the R^2_c value shows that it explains up to 46% variability when random effects are included, (2) In the *comments* model, the variability explained by fixed effects is 66% and including the random effects explains 72% of the variability in the data. This also shows that random effects play a bigger role in the *time* model than in the *comments* model. This difference can also be observed in TABLE VIII. where the variance explained by random effects for the *time* model is considerably higher than the *comments* model.

TABLE VIII. RANDOM EFFECTS RESULTS OF OUR LINEAR MIXED MODELS

	Outcome: Log (Time to close)		**Outcome: Log (#Comments)**	
Groups	***Variance (Std. dev)***	***Corr.***	***Variance (Std. dev)***	***Corr.***
Repository (Intercept)	1.933 (1.390)		0.018 (0.136)	
Repository (Is External)	0.287 (0.536)	-0.66	0.013 (0.113)	0.30
Reporter (Intercept)	1.515 (1.230)		0.019 (0.139)	
Reporter (Is External)	0.374 (0.612)	-0.75	0.022 (0.146)	0.24
Residual	9.138 (3.023)		0.271 (0.521)	
No. of Obs: 53809, groups: repository, 3080; reporter, 6756				

The key takeaway from TABLE VIII. is that the large negative correlations (-0.66, -.75) between the random slope and intercept for both groups (repository, reporter) in the *time* model indicates that as the baseline time decreases, the coefficient of the predictor "*Is External*" increases. In other words, for repositories where internal problems (baseline) take less time to close, the time taken for external problems is considerably higher than the estimates from fixed effects (TABLE VII.).

Therefore, to summarize, the key results of our model are: (a) External problems take about **34% longer** to resolve than internal problems, (b) External problems take relatively longer to resolve in repositories where internal problems have a low average time to close, (c) External problems require **26% more discussions** (comments) than internal problems.

These results quantitatively establish that not only do external problems form a significant proportion but are also considerably difficult to deal with compared to internal problems.

V. DISCUSSIONS

A. *Implications for Software Practice*

Measuring the extent and cost of dependency management problems has several practical implications. First, our approach can help project managers evaluate dependency management solutions by periodically tracking their coordination costs. Moreover, knowing the coordination costs of a particular upstream dependency can help developer make decisions such as removing or migrating to different dependencies [10]. Second, prior research has shown that developers often look for certain signals [80] in a project such as the presence of high status participants, before deciding to contribute to or take a dependency on the project. Therefore, coordination costs can be one form of signal to enable developers to choose a dependency or to help developers decide if a project is worth contributing. Finally, our findings can inform the design of ecosystem policies, for example, we showed that the number of releases of upstream dependencies was positively associated with the number of external problems. One possible implication could be to implement a policy to discourage making frequent releases as it increases coordination costs for downstream projects.

B. *Threats to Validity*

Since we use one particular ecosystem to evaluate coordination costs, our results may not generalize across all software projects and ecosystems. However, our approach is generic and can be adapted to study other ecosystems.

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 [81], there is a potential risk that this filtering process could introduce certain biases.

Although we found a reasonably high F-score of 0.84 for our approach to classify external problems, it is not entirely foolproof and therefore, should be considered when using our results.

Among our predictors of external problems, a few had a small effect size ($< 3\%$), which although was statistically significant may not have a major effect in projects that have very few external problems.

C. *Verifiability*

All sources of data used in this paper are public. We have made available the list of repositories we analyzed, our compiled datasets (for use in our models) and relevant scripts in a public GitHub repository (*arunk054/coordination-costs-icse17*). Broadly, there are three steps involved in replicating our study. First, extract the list of all R package repositories on GitHub that were created before February 2016. Second, extract all issues and pull requests created in those repositories before February 2016 and classify them as *external* using our approach in Sec. III.B. Finally, populate the dataset with all variables (repository, issue and user level) as required by our models. Almost all our variables were computed using data available on GitHub except we use *Crantastic* [82] to identify whether an R package is on CRAN, its version number and number of releases.

VI. CONCLUSION AND FUTURE WORK

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 effort goes into inter-project coordination. In this paper, we estimated the extent and cost of dependency management problems, which we called external problems. We showed using data from Github that 24% of issues in the R ecosystem were external problems; however, taking a dependency on a package accepted into CRAN is associated with fewer external problems than taking a dependency on a package not in CRAN. We demonstrated that external problems incur higher coordination costs than internal problems, measured in terms of the time taken to resolve (34% longer) and number of comments (26% more) in an issue. Future research may study coordination costs of managing dependencies in other ecosystems such as Node.js and Eclipse. There is scope for future work to improve the accuracy of our classification approach, and identify more fine-grained predictors of external problems. Our findings have 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 ecosystem's large-scale collaboration.

REFERENCES

[1] C. Williams, "How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript," 2016. [Online]. Available: http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/. [Accessed: 18-Aug-2016].

[2] 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.

[3] R. D. Banker and R. J. Kauffman, "Reuse and Productivity in Integrated Computer-aided Software Engineering: An Empirical Study," *MIS Q.*, vol. 15, no. 3, pp. 375–401, Oct. 1991.

[4] S. Haefliger, G. von Krogh, and S. Spaeth, "Code Reuse in Open Source Software," *Manage. Sci.*, vol. 54, no. 1, pp. 180–193, 2008.

[5] M. Claes, T. Mens, R. Di Cosmo, and J. Vouillon, "A Historical Analysis of Debian Package Incompatibilities," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 212–223.

[6] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Why Do Software Packages Conflict?," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, 2012, pp. 141–150.

[7] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empir. Softw. Eng.*, pp. 1–47, 2015.

[8] J. Ooms, "Possible Directions for Improving Dependency Versioning in {R}," *CoRR*, vol. abs/1303.2, 2013.

[9] B. E. Cossette and R. J. Walker, "Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, p. 55:1--55:11.

[10] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A Study of Library Migration in Java Software," *CoRR*, vol. abs/1306.6, 2013.

[11] "Greenkeeper." [Online]. Available: https://greenkeeper.io/. [Accessed: 19-Aug-2016].

[12] "Gemnasium." [Online]. Available: https://gemnasium.com/. [Accessed: 19-Aug-2016].

[13] R. Grinter, "From local to global coordination: lessons from software reuse," *Proc. 2001 Int. ACM Siggr. Conf. Support. Gr. Work*, pp. 144–153, 2001.

[14] J. Ossher, S. Bajracharya, and C. Lopes, "Automated dependency resolution for open source software," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, pp. 130–140.

[15] C. R. B. de Souza and D. F. Redmiles, "An Empirical Study of Software Developers' Management of Dependencies and Changes," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 241–250.

[16] M. Lungu, R. Robbes, and M. Lanza, "Recovering Inter-project Dependencies in Software Ecosystems," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 309–312.

[17] D. Le Berre and P. Rapicault, "Dependency Management for the Eclipse Ecosystem: Eclipse P2, Metadata and Resolution," in *Proceedings of the 1st International Workshop on Open Component Ecosystems*, 2009, pp. 21–30.

[18] W. Frakes and C. Terry, "Software Reuse: Metrics and Models," *ACM Comput. Surv.*, vol. 28, no. 2, pp. 415–435, Jun. 1996.

[19] A. Mili, S. F. Chmiel, R. Gottumukkala, and L. Zhang, "An Integrated Cost Model for Software Reuse," in *Proceedings of the 22Nd International Conference on Software Engineering*, 2000, pp. 157–166.

[20] D. M. German, B. Adams, and A. E. Hassan, "The Evolution of the R Software Ecosystem," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, 2013, pp. 243–252.

[21] K. Krippendorff, *Content analysis: An introduction to its methodology*. Sage, 2012.

[22] D. Lambert, "Zero-Inflated Poisson Regression, with an Application to Defects in Manufacturing," *Technometrics*, vol. 34, no. 1, pp. 1–14, 1992.

[23] P. D. Allison and R. Waterman, "Fixed-Effects Negative Binomial Regression Models," in *Sociology Methodology*, 2002, vol. 32, pp. 247–265.

[24] D. Bates, M. Mächler, B. Bolker, and S. Walker, "Fitting Linear Mixed-Effects Models Using {lme4}," *J. Stat. Softw.*, vol. 67, no. 1, pp. 1–48, 2015.

[25] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 242–251.

[26] 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.

[27] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empir. Softw. Eng.*, vol. 12, no. 5, pp. 471–516, 2007.

[28] B. Barnes, T. Durek, J. Gaffney, and A. Pyster, "Software Reuse: Emerging Technology," W. Tracz, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 77–88.

[29] G. von Krogh, S. Spaeth, and K. R. Lakhani, "Community, joining, and specialization in open source software innovation: a case study," *Res. Policy*, vol. 32, no. 7, pp. 1217–1241, 2003.

[30] J. C. Knight and M. F. Dunn, "Software quality through domain-;driven certification," *Ann. Softw. Eng.*, vol. 5, no. 1, pp. 293–315, 1998.

[31] I. Crnkovic, "Component-based software engineering — new challenges in software development," *Softw. Focus*, vol. 2, no. 4, pp. 127–133, 2001.

[32] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[33] C. R. B. de Souza, D. Redmiles, and P. Dourish, "'Breaking the Code', Moving Between Private and Public Work in Collaborative Software Development," in *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, 2003, pp. 105–114.

[34] I. van den Berk, S. Jansen, and L. Luinenburg, "Software Ecosystems: A Software Ecosystem Strategy Assessment Model," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, 2010, pp. 127–134.

[35] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "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)*, 2016.

[36] W. Wang and M. W. Godfrey, "Detecting API Usage Obstacles: A Study of iOS and Android Developer Questions," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 61–64.

[37] A. Decan, T. Mens, M. Claes, and P. Grosjean, "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)*, 2016, vol. 1, pp. 493–504.

[38] M. S. Zargar, "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*, 2013.

[39] E. H. Trainer, C. Chaihirunkarn, A. Kalyanasundaram, and J. D. Herbsleb, "From Personal Tool to Community Resource: What's the Extra Work and Who Will Do It?," in *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work &#38; Social Computing*, 2015, pp. 417–430.

[40] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools," in *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, 2006, pp. 353–362.

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

[42] Y. Ye, K. Nakakoji, and Y. Yamamoto, "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*, 2007, pp. 152–169.

[43] M. Cataldo, M. Bass, J. D. Herbsleb, and L. Bass, "On Coordination Mechanisms in Global Software Development," in *International Conference on Global Software Engineering (ICGSE 2007)*, 2007, pp. 71–80.

[44] J. D. Herbsleb and A. Mockus, "An Empirical Study of Speed and Communication in Globally Distributed Software Development," *IEEE Trans. Softw. Eng.*, vol. 29, no. 6, pp. 481–494, Jun. 2003.

[45] K. Blincoe, G. Valetto, and D. Damian, "Do All Task Dependencies Require Coordination? The Role of Task Properties in Identifying Critical Coordination Needs in Software Projects," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 213–223.

[46] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of Social and Technical Factors for Evaluating Contribution in GitHub," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 356–366.

[47] R Development Core Team, "R: A Language and Environment for Statistical Computing." Vienna, Austria, 2008.

[48] S. Voulgaropoulou, G. Spanos, and L. Angelis, "Analyzing Measurements of the R Statistical Open Source Software," in *Software Engineering Workshop (SEW), 2012 35th Annual IEEE*, 2012, pp. 1–10.

[49] G. Robles, J. M. González-Barahona, C. Cervigón, A. Capiluppi, and D. Izquierdo-Cortázar, "Estimating Development Effort in Free/Open Source Software Projects by Mining Software Repositories: A Case Study of OpenStack," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 222–231.

[50] "Github Linguist." [Online]. Available: https://github.com/github/linguist. [Accessed: 18-Aug-2016].

[51] G. Gousios, M. Pinzger, and A. van Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 345–355.

[52] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, and Y. L. Traon, "Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 188–197.

[53] E. H. Trainer, A. Kalyanasundaram, C. Chaihirunkarn, and J. D. Herbsleb, "How to Hackathon: Socio-technical Tradeoffs in Brief, Intensive Collocation," in *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, 2016, pp. 1118–1130.

[54] "Creating and highlighting code blocks." [Online]. Available: https://help.github.com/articles/creating-and-highlighting-code-blocks/. [Accessed: 18-Aug-2016].

[55] A. Gelbukh, *Computational Linguistics and Intelligent Text Processing: 12th International Conference, CICLing 2011, Tokyo, Japan, February 20-26, 2011. Proceedings*. Springer, 2011.

[56] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods for Software Engineering Research," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London: Springer London, 2008, pp. 285–311.

[57] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 557–572, Jul. 1999.

[58] I. Abal, C. Brabrand, and A. Wasowski, "42 Variability Bugs in the Linux Kernel: A Qualitative Analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 421–432.

[59] K. Blincoe, G. Valetto, and D. Damian, "Uncovering Critical Coordination Requirements Through Content Analysis," in *Proceedings of the 2013 International Workshop on Social Software Engineering*, 2013, pp. 1–4.

[60] J. Corbin and J. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 3rd ed. Sage, 2008.

[61] E. Raymond, "The cathedral and the bazaar," *Knowledge, Technol. {&} Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[62] K. Crowston and J. Howison, "Hierarchy and centralization in free and open source software team communications," *Knowledge, Technol. {&} Policy*, vol. 18, no. 4, pp. 65–85, 2006.

[63] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov, "The Sky is Not the Limit: Multitasking Across GitHub Projects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 994–1005.

[64] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 364–373.

[65] C. F. Dormann, J. Elith, S. Bacher, C. Buchmann, G. Carl, G. Carré, J. R. G. Marquéz, B. Gruber, B. Lafourcade, P. J. Leitão, T. Münkemüller, C. McClean, P. E. Osborne, B. Reineking, B. Schröder, A. K. Skidmore, D. Zurell, and S. Lautenbach, "Collinearity: a review of methods to deal with it and a simulation study evaluating their performance," *Ecography (Cop.).*, vol. 36, no. 1, pp. 27–46, 2013.

[66] Q. H. Vuong, "Likelihood Ratio Tests for Model Selection and Non-nested Hypotheses," *Econometrica*, vol. 57, no. 2, pp. 307–333, 1989.

[67] S. Jackman, "{pscl}: Classes and Methods for {R} Developed in the Political Science Computational Laboratory, Stanford University." Stanford, California, 2015.

[68] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Taylor & Francis, 2013.

[69] A. F. Hayes, "Beyond Baron and Kenny: Statistical Mediation Analysis in the New Millennium," *Commun. Monogr.*, vol. 76, no. 4, pp. 408–420, 2009.

[70] H. Schielzeth and W. Forstmeier, "Conclusions beyond support: overconfident estimates in mixed models," *Behav. Ecol.*, vol. 20, no. 2, pp. 416–420, 2009.

[71] D. J. Barr, "Random effects structure for testing interactions in linear mixed-effects models," *Frontiers in Psychology*, vol. 4. 2013.

[72] "Autolinked references and URLs." [Online]. Available: https://help.github.com/articles/autolinked-references-and-urls/. [Accessed: 18-Aug-2016].

[73] 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.

[74] A. Kuznetsova, P. Bruun Brockhoff, and R. Haubo Bojesen Christensen, "lmerTest: Tests in Linear Mixed Effects Models." 2016.

[75] F. E. Satterthwaite, "Synthesis of variance," *Psychometrika*, vol. 6, no. 5, pp. 309–316, 1941.

[76] "VIF Mixed Effect Models." [Online]. Available: https://github.com/aufrank/R-hacks/blob/master/mer-utils.R. [Accessed: 18-Aug-2016].

[77] F. E. Harrell Jr, "rms: Regression Modeling Strategies." 2016.

[78] S. Nakagawa and H. Schielzeth, "A general and simple method for obtaining R2 from generalized linear mixed-effects models," *Methods Ecol. Evol.*, vol. 4, no. 2, pp. 133–142, 2013.

[79] K. Bartoń, "MuMIn: Multi-Model Inference." 2016.

[80] J. Tsay, L. Dabbish, and J. D. Herbsleb, "Social media in transparent work environments," in *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, 2013, pp. 65–72.

[81] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The Promises and Perils of Mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 92–101.

[82] "It's crantastic." [Online]. Available: http://crantastic.org/. [Accessed: 18-Aug-2016].