Why Do Developers Use Trivial Packages? An Empirical Case Study on *npm*

Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab
Data-driven Analysis of Software (DAS) Lab
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
{rab_abdu,o_nourry,s_alweha,s_mujahi,eshihab}@encs.concordia.ca

ABSTRACT

Code reuse is traditionally seen as good practice. Recent trends have pushed the concept of code reuse to an extreme, by using packages that implement simple and trivial tasks, which we call 'trivial packages'. A recent incident where a trivial package led to the breakdown of some of the most popular web applications such as Facebook and Netflix made it imperative to question the growing use of trivial packages.

Therefore, in this paper, we mine more than 230,000 *npm* packages and 38,000 JavaScript applications in order to study the prevalence of trivial packages. We found that trivial packages are common and are increasing in popularity, making up 16.8% of the studied *npm* packages. We performed a survey with 88 Node.js developers who use trivial packages to understand the reasons and drawbacks of their use. Our survey revealed that trivial packages are used because they are perceived to be well implemented and tested pieces of code. However, developers are concerned about maintaining and the risks of breakages due to the extra dependencies trivial packages introduce. To objectively verify the survey results, we empirically validate the most cited reason and drawback and find that, contrary to developers' beliefs, only 45.2% of trivial packages even have tests. However, trivial packages appear to be 'deployment tested' and to have similar test, usage and community interest as non-trivial packages. On the other hand, we found that 11.5% of the studied trivial packages have more than 20 dependencies. Hence, developers should be careful about which trivial packages they decide to use.

CCS CONCEPTS

•**Software and its engineering → Software libraries and repositories;** *Software maintenance tools;*

KEYWORDS

JavaScript; Node.js; Code Reuse; Empirical Studies

ACM Reference format:
Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on *npm*. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
DOI: 10.1145/3106237.3106267

1 INTRODUCTION

Code reuse is often encouraged due to its multiple benefits. In fact, prior work showed that code reuse can reduce the time-to-market, improve software quality and boost overall productivity [3, 32, 37]. Therefore, it is no surprise that emerging platforms such as Node.js encourage reuse and do everything possible to facilitate code sharing, often delivered as packages or modules that are available on package management platforms, such as the Node Package Manager (*npm*) [7, 39].

However, it is not all good news. There are many cases where code reuse has had negative effects, leading to an increase in maintenance costs and even legal action [2, 29, 35, 41]. For example, in a recent incident code reuse of a Node.js package called left-pad, which was used by Babel, caused interruptions to some of the largest Internet sites, e.g., Facebook, Netflix, and Airbnb. Many referred to the incident as the case that 'almost broke the Internet' [33, 45]. That incident lead to many heated discussions about code reuse, sparked by David Haney's blog post: *"Have We Forgotten How to Program?"* [26].

While the real reason for the left-pad incident was that *npm* allowed authors to unpublish packages (a problem which has been resolved [40]), it raised awareness of the broader issue of taking on dependencies for trivial tasks that can be easily implemented [26]. Since then, there have been many discussions about the use of trivial packages. Loosely defined, *a trivial package is a package that contains code that a developer can easily code him/herself and hence, is not worth taking on an extra dependency for.* Many developers agreed with Haney's position, which stated that every serious developer knows that 'small modules are only nice in theory' [8], suggesting that developers should implement such functions themselves rather than taking on dependencies for trivial tasks. Other work showed that *npm* packages tend to have a large number of dependencies [13, 14] and highlighted that developers need to use caution since some dependencies can grow exponentially [4]. In fact, in our dataset, we found that more than 11% of the trivial packages have more than 20 dependencies.

So, the million dollar question is "why do developers resort to using a package for trivial tasks, such as checking if a variable is an array?" At the same time, other questions regarding how prevalent trivial packages are and what the potential drawbacks of using these trivial packages remain unanswered. Therefore, we performs an empirical study involving more than 230,000 *npm* packages and 38,000 JavaScript applications to better understand why developers resort to using trivial packages. Our empirical study is qualitative in nature and is based on survey results from 88 Node.js developers. We also quantitatively validate the most commonly developer-cited reason and drawback related to the use of trivial packages.

Since, to the best of our knowledge, this is the first study to examine why developers use trivial packages, we first propose a definition of what constitutes a trivial package, based on feedback from JavaScript developers. We also examine how prevalent trivial packages are in *npm* and how widely they are used in Node.js applications. Our findings indicate that:

Trivial packages are common and popular. Of the 231,092 *npm* packages in our dataset, 16.8% of them are trivial packages. Moreover, of the 38,807 Node.js applications on GitHub, 10.9% of them directly depend on one or more trivial packages.

Most developers do not consider the use of trivial packages as bad practice. In our survey of the 88 JavaScript developers, 57.9% of them said they do not consider the use of trivial packages as bad practice, whereas only 23.9% consider it to be a bad practice. This finding shows that there is not a clear consensus on the issue of trivial package use.

Trivial packages provide well implemented and tested code and increase productivity. Developers believe that trivial packages provide them with well implemented/tested code and increase productivity. At the same time, the increase in dependency overhead and the risk of breakage of their applications are the two most cited drawbacks.

Developers need to be careful which trivial packages they use. Our empirical findings show that many trivial packages have their own dependencies. In fact, we found that 43.7% of trivial packages have at least one dependency and 11.5% of trivial packages have more than 20 dependencies.

In addition to the aforementioned findings, our study provides the following key contributions:

- We provide a way to quantitatively determine trivial packages.
- To the best of our knowledge, this is the first study to examine the prevalence, reasons for and drawbacks of using trivial packages in Node.js applications. Our study is also one of the largest studies on JavaScript applications, involving a survey of more than 80 JavaScript developers, 231,092 *npm* packages and 38,807 Node.js applications.
- We perform an empirical study to validate the most commonly cited reasons for and drawbacks of using trivial packages in our developer survey.
- We make our dataset of the responses provided by the *npm* developers publicly available. [1]

The paper is organized as follows: Section 2 provides the background and introduces our datasets. Section 3 presents how we determine what a trivial package is. Section 4 examines the prevalence of trivial packages and their use in Node.js applications. Section 5 presents the results of our developer survey, presenting the reasons and perceived drawbacks for developers who use trivial packages. Section 6 presents our empirical validation of the most commonly cited reason for and drawback of using trivial packages. The implications of our findings are noted in section 7. We discuss the related works in section 8, the limitations of our study in section 9, and present our conclusions in section 10.

2 BACKGROUND AND DATASETS

JavaScript is used to write client and server side applications. Its popularity has steadily grown, thanks to popular frameworks such as Node.js and an active developer community [7, 46]. JavaScript projects can be classified into two main categories: *packages* that are used in other projects or *applications* that are used as standalone software. The Node Package Manager (*npm*) provides tools to manage Node.js packages. *npm* is the official package manager for Node.js and its registry contains more than 250,000 packages [25].

To perform our study, we gather two datasets from two sources. We obtain Node.js packages from the *npm* registry and applications that use *npm* packages from GitHub.

Packages: Since we are interested in examining the impact of 'trivial' packages, we mined the latest version of all the Node.js packages from *npm* as of May 5, 2016. For each package we obtained its source code from GitHub. In some cases, the package publisher did not provide a GitHub link, in which case we obtained the source code directly from *npm*. In total, we mined 252,996 packages.

Applications: We also want to examine the use of the packages in JavaScript applications. Therefore, we mined all of the Node.js applications on GitHub. To ensure that we are indeed only obtaining the applications from GitHub, and not *npm* packages, we compare the URL of the GitHub repositories to all of the URLs we obtained from *npm* for the packages. If a URL from GitHub was also in *npm*, we flagged it as being an *npm* package and removed it from the application list. To determine that an application uses *npm* packages, we looked for the 'package.json' file, which specifies (amongst others) the *npm* package dependencies used by the application.

To eliminate dummy applications that may exist in GitHub, we choose non-forked applications with more than 100 commits and more than 2 developers. Similar filtering criteria were use in prior work by Kalliamvakou *et al.* [31]. In total, we obtained 115,621 JavaScript applications and after removing applications that did not use the *npm* platform, we were left with 38,807 applications.

3 WHAT ARE TRIVIAL PACKAGES ANYWAY?

Although what a trivial package is has been loosely defined in the past (e.g., in blogs [27, 28]), we want a more precise and objective way to determine trivial packages. To determine what constitutes a trivial package, we conducted a survey, where we asked participants what they considered to be a trivial package and what indicators they used to determine if a package is trivial or not. We devised an online survey that presented the source code of 16 randomly selected Node.js packages that range in size between 4 - 250 JavaScript lines of code (LOC). Participants were asked to 1) indicate if they thought the package was trivial or not and 2) specify what indicators they use to determine a trivial package. We opted to

[1] http://das.encs.concordia.ca/publications/npm-survey-data/

limit the size of the Node.js packages in the survey to a maximum of 250 JavaScript LOC since we did not want to overwhelm the participants with the review of excessive amounts of code.

We asked the survey participants to indicate trivial packages from the list of Node.js packages provided. We provided the survey participants with a loose definition of what a trivial package is, i.e., a package that contains code that they can easily code themselves and hence, is not worth taking on an extra dependency for. Figure 1 shows an example of a trivial package, called *is-Positive*, which simply checks if a number is positive. The survey questions were divided into three parts: 1) questions about the participant's development background, 2) questions about the classification of the provided Node.js packages and 3) questions about what indicators the participant would use to determine a trivial package. We sent the survey to 22 developers and colleagues that were familiar with JavaScript development and received a total of 12 responses.

```
module.exports = function (n) {
  return toString.call(n) === '[object Number]' && n > 0;
};
```

Figure 1: Package is-Positive on *npm*

Participants Background and Experience. Of the 12 respondents, 2 are undergraduate students, 8 are graduate students and 2 are professional developers. Ten of the 12 respondents have at least 2 years of JavaScript experience and half of the participants have been developing with JavaScript for more than five years.
Survey Responses. We asked participants to list what indicators they use to determine if a package is trivial or not and to indicate all the packages that they considered to be trivial. Of the 12 participants, 11 (92%) state that the complexity of the code and 9 (75%) state that size of the code are indicators they use to determine a trivial package. Another 3 (20%) mentioned that they used code comments and other indicators (e.g., functionality) to indicate if a package is trivial or not. Since it is clear that size and complexity are the most common indicators of trivial packages, we use these two measures to determine trivial packages. It should be mentioned that participants could provide more than 1 indicator, hence the percentages above sum to more than 100%.

Next, we analyze all of the packages that were marked as trivial. In total, we received 69 votes for the 16 packages. We ranked the packages in ascending order, based on their size, and tallied the votes for the most voted packages. We find that 79% of the votes consider packages that are less than 35 lines of code to be trivial. We also examine the complexity of the packages using McCabe's cyclomatic complexity, and find that 84% of the votes marked packages that have a total complexity value of 10 or lower to be trivial. It is important to note that although we provide the source code of the packages to the participants, we do not explicitly provide the size or the complexity of the packages to the participants, so they are not biased by any metrics, i.e., size or complexity, in their classification.

Based on the aforementioned findings, we used the two indicators JavaScript LOC $\leq$ 35 and complexity $\leq$ 10 to determine trivial packages in our dataset. Hence, we define trivial packages as $\left\{X_{LOC} \leq 35 \cap X_{Complexity} \leq 10\right\}$, where X_{LOC} represents the JavaScript LOC and $X_{Complexity}$ represents McCabe's cyclomatic complexity of package X. Although we use the aforementioned measures to determine trivial packages, we do not consider this to be the only possible way to determine trivial packages.

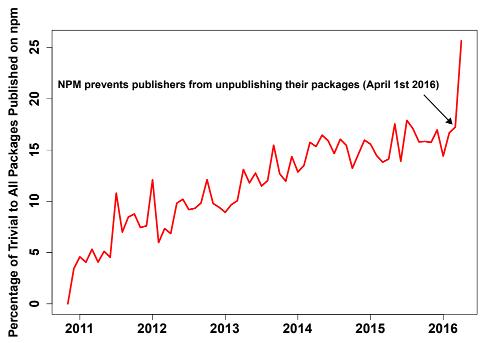


Figure 2: Percentage of Published Trivial Packages on *npm*.

Our survey indicates that size and complexity are commonly used measures to determine if a package is trivial. Based on our analysis, packages that have $\leq$ 35 JavaScript LOC and a McCabe's cyclomatic complexity $\leq$ 10 are considered to be trivial.

4 HOW PREVALENT ARE TRIVIAL PACKAGES?

In this section, we want to know how prevalent trivial packages are. We examine prevalence from two aspects: the first aspect is from *npm*'s perspective, where we are interested in knowing how many of the packages on *npm* are trivial. The second aspect considers the use of trivial packages in JavaScript applications.

4.1 How Many of npm's Packages are Trivial?

We use the two measures, LOC and complexity, to determine trivial packages, which we now use to quantify the number of trivial packages in our dataset. Our dataset contained a total of 252,996 *npm* packages. For each package, we calculated the number of JavaScript code lines and removed packages that had zero LOC, which removed 21,904 packages. This left us with a final number of 231,092 packages. Then, for each package, we removed test code since we are mostly interested in the actual source code of the packages. To identify and remove the test code, similar to prior work [22, 44, 48], we look for the term "test" (and its variants) in the file names and file paths.

Out of the 231,092 *npm* packages we mined, 38,845 (16.8%) packages are trivial packages. In addition, we examined the growth of trivial packages in *npm*. Figure 2 shows the percentage of trivial to all packages published on *npm* per month. We see an increasing trend in the number of trivial packages over time and approximately 15% of the packages added every month are trivial packages. We investigated the spike around March 2016 and found that this spike corresponds to the time when *npm* disallowed the un-publishing of packages [40].

npm posts the most depended-upon packages on its website [38]. We measured the number of trivial packages that exist in the top 1,000 most depended-upon packages; we find that 113 of them are trivial packages. This finding shows that trivial packages are not only prevalent and increasing in number, but they are also very popular among developers, making up 11.3% of the 1,000 most depended on *npm* packages.

Trivial packages make up 16.8% of the studied npm packages. Moreover, the proportion of trivial packages is increasing and trivial packages make up 11.3% of the top 1,000 most depended on npm packages.

4.2 How Many Applications Depend on Trivial Packages?

Just because trivial packages exist on *npm*, it does not mean that they are actually being used. Therefore, we also examine the number of applications that use trivial packages. To do so, we examine the package.json file, which contains all the dependencies that an application installs from *npm*. However, in some cases, an application may install a package but not use it. To avoid counting such instances, we parse the JavaScript code of all the examined applications and use regular expressions to detect the require dependency statements, which indicates that the application actually uses the package in its code[2]. Finally, we measured the number of packages that are trivial in the set of packages used by the applications. Note that we only consider *npm* packages since it is the most popular package manager for Node.js packages and other package managers only manage a subset of packages (e.g., Bower [9] only manages front-end/client-side frameworks, libraries and modules). We find that of the 38,807 applications in our data set, 4,256 (10.9%) directly depend on at least one trivial package.

Of the 38,807 Node.js applications in our dataset, 10.9% of them depend on at least one trivial package.

5 SURVEY RESULTS

We surveyed Node.js developers to understand the reasons for and the drawbacks of using trivial packages. We use a survey because it allows us to obtain first-hand information from the developers who use these trivial packages. In order to select the most relevant participants, we sent out the survey to developers who use trivial packages. We used Git's `pickaxe` command on the lines that contain the required dependency statements in the applications; a procedure that provided us with the email and name of the developer who introduced the trivial package dependency.

Survey Participants. To mitigate the possibility of introducing misunderstood or misleading questions, we initially sent the survey to two JavaScript developers and incorporated their minor suggestions to improve the survey. Next, we sent the survey to 1,055 developers from 1,696 applications. To select the developers, we ranked them based on the number of trivial packages they use. We then took a sample of 600 developers that use trivial packages the most, and another 600 of those that indicated the least use of trivial packages. The survey was emailed to the 1,200 selected developers, however, since some of the emails were returned for various reasons (e.g., the email account does not exist anymore, etc.), we could only reach 1,055 developers.

The survey listed the trivial package and the application that we detected the trivial package in. We received 88 responses to our survey, which translates to a response rate of 8.3%. Our survey response rate is in line with, and even higher, than the typical 5% response rate reported in questionnaire-based software engineering surveys [42]. Of the 88 respondents, 83 of them identified as developers working either in industry (68) or as a full time independent developers (15). The remaining 5 identified as being a casual developers (2) or other (3), including one student and two developers working in executive positions at *npm*. As for the development experience of the survey respondents, the majority (67) of the respondents have more than 5 years of experience, 14 have between 3-5 years and 7 have 1-3 years of experience. The fact that most of the respondents are experienced JavaScript developers gives us confidence in our survey responses.

5.1 Do Developers Consider Trivial Packages Harmful?

The first question of our survey to the participants is: "Do you consider the use of trivial packages as bad practice?" The reason to ask this question so bluntly is that it allows us to gauge, in a very deterministic way, how the Node.js developers felt about the issue of using trivial packages. We provided three possible replies, Yes, No or Other in which case they were provided with a text box to elaborate. Of the 88 participants, 51 (57.9%) stated that they do NOT consider the use of trivial packages as bad practice. Another 21 (23.9%) stated that they indeed think that using trivial package is a bad practice. The remaining 16 (18.2%) stated that it really depends on the circumstances, such as the time available, how critical a piece of code is, and if the package used has been thoroughly tested.

Most of the surveyed developers (57.9%) do NOT believe that using trivial packages is a bad practice.

5.2 Why Do Developers Use Trivial Packages?

While we have answered the question as to whether developers think using trivial packages is a bad practice, what we are most interested in is why do developers resort to using trivial packages and what do they view as the drawbacks of using trivial packages. Therefore, the second part of the survey asks participants to list the reasons why they resort to using trivial packages. To ensure that we do not bias the responses of the developers, the answer fields for these questions were in free-form text, i.e., no predetermined suggestions were provided. After gathering all of the responses, we grouped and categorized the responses in a two-phase iterative process. In the first phase, the first two authors carefully read the participant's answers and came up with a number of categories that the responses fell under. Next, they discussed their groupings and agreed on the extracted categories. Whenever they failed to agree on a category, a third author was asked to help break the tie. Once all of the categories were decided, the same two authors went through all the answers again and classified them into their respective categories. For the majority of the cases, the two authors agreed on most categories and the classifications of the responses. To measure the agreement between the two authors, we used Cohen's Kappa coefficient [10]. The Cohen's Kappa coefficient has been used to evaluate inter-rater agreement levels for categorical scales, and provides the proportion of agreement corrected for chance. The resulting coefficient is scaled to range between -1 and

[2]Note that if a package is required in the application, but does not exist, it will break the application.

Table 1: Reasons for using trivial packages.

Reason	Description	#Resp.	%
Well implemented & tested	Participants state that trivial packages are effectively implemented and tested.	48	54.6%
Increased productivity	Trivial packages reduce the time needed to implement existing source code.	42	47.7%
Well maintained code	It eases source code maintenance, since other developers maintain the trivial package.	8	9.1%
Improved readability & reduced complexity	Using trivial packages improve the source code quality in terms of readability and reduce complexity.	8	9.1%
Better performance	Trivial packages improve the performance of web applications compared to the use of large frameworks.	3	3.4%
No reason	-	7	8.0%

+1, where a negative value means less than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement [18]. In our categorization, the level of agreement measured between the authors was of +0.90, which is considered to be an excellent inter-rater agreement.

Table 1 shows the five reasons for using trivial packages, as reported by our survey respondents; another category was used to group the 'no reason' responses. Table 1 presents the different reasons, a description of each category and its frequency. These reasons are listed below, in order of their popularity:

R1. Well implemented & tested (54.6%): The most cited reason for using trivial packages is that they provide well implemented and tested code. More than half of the responses mentioned this reason. In particular, although it may be easy for developers to code these trivial packages themselves, it is more difficult to make sure that all the details are addressed, e.g., one needs to carefully consider all edge cases. Some example responses that mention these issues are stated by participants P68 and P4, who cite their reasons for using trivial packages as follows: P68: *"Tests already written, a lot of edge cases captured [...]."* & P4: *"There may be a more elegant/efficient/correct/cross-environment-complatible solution to a trivial problem than yours"*.

R2. Increased productivity (47.7%): The second most cited reason is the improved productivity that using trivial packages enables. Trivial tasks or not, writing code on your own requires time and effort, hence, many developers view the use of trivial packages as a way to boost their productivity. In particular, early on in a project, a developer does not want to worry about small details, they would rather focus their efforts on implementing the more difficult tasks. For example, participants P13 and P27 state: P13: *"[...] and it does save time to not have to think about how best to implement even the simple things."* & P27: *"Don't reinvent the wheel! if the task has been done before."*. The aforementioned are clear examples of how developers would rather not code something, even if it is trivial. Of course, this comes at a cost, which we discuss later.

R3. Well maintained code (9.1%): A less common, but cited reason for using trivial packages is the fact that the maintenance of the code need not to be performed by the developers themselves; in essence, it is outsourced to the community or the contributors of the trivial packages. For example, participant P45 states: *"Also, a highly used trivial package is probable to be well maintained."*. Even tasks such as bug fixes are dealt with by the contributors of the trivial packages, which is very attractive to the users of the trivial packages, as reported by participant P80: *"[...], leveraging feedback from a larger community to fix bugs, etc."*

R4. Improved readability & reduced complexity (9.1%): Participants also reported that using trivial packages improves the readability and reduces the complexity of their code. For example, P34 states: *"immediate clarity of use and readability for other developers for commonly used packages[...]"* & P47 states: *"Simple abstract brings less complexity."*

R5. Better performance (3.4%): A few of the participants stated that using trivial packages improves performance since it alleviates the need for their application to depend on large frameworks. For example, P35 states: *"[...] you do not depend on some huge utility library of which you do not need the most part."*

Only a small percentage (8.0%) of the respondents stated that they do not see a reason to use trivial packages.

The two most cited reasons for using trivial packages are 1) they provide well implemented and tested code and 2) they increase productivity.

5.3 Drawbacks of Using Trivial Packages

In addition to knowing the reasons why developers resort to trivial packages, we wanted to understand the other side of the coin - what they perceive to be the drawbacks of their decision to use these packages. The drawbacks question was part of our survey and we followed the same aforementioned process to analyze the survey responses. In the case of the drawbacks the Cohen's Kappa agreement measure was +0.86, which is considered to be an excellent agreement. Table 2 lists the drawback mentioned by the survey respondents along with a brief description and the frequency of each drawback.

I1. Dependency overhead (55.7%): The most cited drawback of using trivial packages is the increased dependency overhead, e.g., keeping all dependencies up to date and dealing with complex dependency chains, that developers need to bear [7]. This situation is often referred to as 'dependency hell', especially when the trivial packages themselves have additional dependencies. This drawback came through clearly in many comments, for example, P41 states: *"[...] people who don't actively manage their dependency versions could [be] exposed to serious problems [...]"* & P40: *"Hard to maintain a lot of tiny packages"*. Hence, while trivial packages may provide well

Table 2: Drawbacks of using trivial packages.

Drawback	Description	# Resp.	%
Dependency overhead	Using trivial packages results in a dependency mess that is hard to update and maintain.	49	55.7%
Breakage of applications	Depending on a trivial package could cause the application to break if the package becomes unavailable or has a breaking update.	16	18.2%
Decreased performance	Trivial packages decrease the performance of applications, which includes the time to install and build the application.	14	15.9%
Slows development	Finding a relevant and high quality trivial package is a challenging and time consuming task.	11	12.5%
Missed learning opportunities	The practice of using trivial packages leads to developers not learning and experiencing writing code for trivial tasks.	8	9.1%
Security	Using trivial packages can open a door for security vulnerability.	7	8.0%
Licensing issues	Using trivial packages could cause licensing conflicts.	3	3.4%
No drawbacks	-	7	8.0%

implemented/tested code and improve productivity, developers are clearly aware that the management of the additional dependencies is something they need to deal with.

I2. Breakage of applications (18.2%): Developers also worry about the potential breakage of their application due to a specific package or version becoming unavailable. For example, in the left-pad issue, the main reason for the breakage was the removal of left-pad, P4 states: *"Obviously the whole 'left-pad crash' exposed an issue"*. However, since that incident, *npm* has disabled the possibility of a package to be removed [40]. Although disallowing the removal solves part of the problem, packages can still be updated, which may break an application. For a non-trivial package, it may be worth it to take the risk, however, for trivial packages, it may not be worth taking such a risk.

I3. Decreased performance (15.9%): This issue is related to the dependency overhead drawback. Developers mentioned that incurring the additional dependencies slowed down the build time and increased application installation times. For example, P64 states: *"Too many metadata to download and store than a real code."* & P34 states: *"[...], slow installs; can make project noisy and unintuitive by attempting to cobble together too many disparate pieces instead of more targeted code."*. As mentioned earlier, in some cases it is not just the fact that the trivial package adds a dependency, but in some cases the trivial package itself depends on additional packages, which negatively impacts performance even further.

I4. Slows development (12.5%): In some cases, the use of trivial packages may actually have a reverse effect and slow down development. For example, as P23 and P15 state: P23: *"Can actually slow the team down as, no matter how trivial a package, if a developer hasn't required it themselves they will have to read the docs in order to double check what it does, rather than just reading a few lines of your own source."* & P15: *"[...], we have the problem of locating packages that are both useful and "trustworthy" [...]"*; it can be difficult to find a relevant and trustworthy package. Even if others try to build on your code, it is much more difficult to go fetch a package and learn it, rather than read a few lines of your code.

I5. Missed learning opportunities (9.1%): In certain cases, the use of these trivial packages is seen as a missed learning opportunity for developers. For example, P24 states: *"Sometimes people forget how to do things and that could lead to a lack of control and knowledge of the language/technology you are using"*. This is a clear example of where just using a package, rather than coding the solution yourself, will lead to less knowledge about the code base.

I6. Security (8.0%): In some cases the trivial packages may have security flaws that make the application more vulnerable. This is an issue pointed out by a few developers, for example, as P15 mentioned earlier, it is difficult to find packages that are trustworthy. P57 also mentions: *"If you depend on public trivial packages then you should be very careful when selecting packages for security reasons"*. As in the case of any dependency one takes on, there is always a chance that a security vulnerability could be exposed in one of these packages.

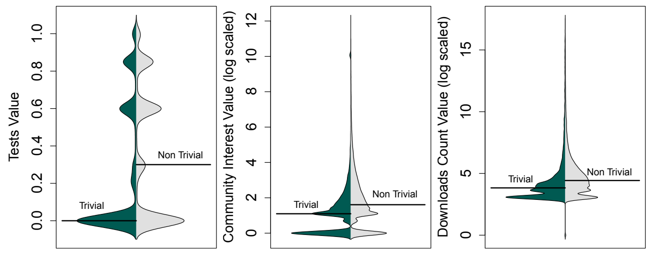
I7. Licensing issues (3.4%): In some cases, developers are concerned about potential licensing conflicts that trivial packages may cause. For example, P73 states: *"[...], possibly license-issues"*, P62: *"[...], there is a risk that the 'trivial' package might be licensed under the GPL must be replaced anyway prior to shipping."*

There were also 8% of the responses that stated they do not see any drawbacks with using trivial packages.

The two most cited drawbacks of using trivial packages are 1) they increase dependency overhead and 2) they may break their applications due to a package or a specific version becoming unavailable or incompatible.

6 PUTTING DEVELOPER PERCEPTION UNDER THE MICROSCOPE

The developer survey provided us with great insights on why developers use trivial packages and what they perceive to be their drawbacks. However, whether there is empirical evidence to support their perceptions remains unexplored. Thus, we examine the most commonly cited reason for using trivial packages, i.e., the fact that trivial packages are well tested, and drawback, i.e., the impact of additional dependencies, based on our findings in Section 5.



(a) Tests **(b) Community Interest** **(c) Download Count**

Figure 3: Distribution of Tests, Community Interest and Download Count Metrics.

6.1 Examining the 'Well Tested' Perception

As shown in Table 1, 54.6% of the responses indicate that they use trivial packages since they are well implemented and tested. And, the developers have good reasons to believe so. *npm* requires that developers provide a test script name with the submission of their packages (listed in the package.json file). In fact, 81.2% (31,521 out of 38,845) of the trivial packages in our dataset have some test script name listed. However, since developers can provide any script name under this field, it is difficult to know if a package is *actually* tested.

We examine whether a package is really well tested and implemented from two aspects; first, we check if a package has tests written for it. Second, since in many cases, developers consider packages to be 'deployment tested', we also consider the usage of a package as an indicator of it being well tested and implemented [47]. To carefully examine whether a package is really well tested and implemented, we use the *npm* online search tool (known as *npms* [11]) to measure various metrics related to how well the packages are tested, used and valued. To provide its ranking of the packages, *npms* mines and calculates a number of metrics based on development (e.g., tests) and usage (e.g., no. of downloads) data. We use three metrics measured by *npms* to validate the 'well tested and implemented' perception of developers, which are[3]:

1) Tests: considers the tests' size, coverage percentage and build status for a project. We looked into the *npms* source code and find that the Tests metric is calculated as: $testsSize * 0.6 + buildStatus * 0.25 + coveragePercentage * 0.15$. We use the Tests metric to determine if a package is tested and how trivial packages compare to non-trivial packages in terms of how well tested they are. One example that motives us to investigate how well tested a trivial package is the response by P68, who says: *"Tests already written, a lot edge cases captured [...]."*.

2) Community interest: evaluates the community interest in the packages, using the number of stars on GitHub & *npm*, forks, subscribers and contributors. Once again, we find through the source code of *npms* that Community interest is simply the sum of the aforementioned metrics, measured as: $starsCount + forksCount + subscribersCount + contributorsCount$. We use this metric to compare how interested the community is in trivial and non-trivial packages. We measure the community interest since developers view the importance of the trivial packages as evidence of its quality as stated by P56, who says: *"[...] Using an isolated module that is well-tested and vetted by a large community helps to mitigate the chance of small bugs creeping in."*.

3) Download count: measures the mean downloads for the last three months. Again, the number of downloads of a package is often viewed as an indicator of the package's quality; as P61 mentions: *"this code is tested and used by many, which makes it more trustful and reliable."*.

[3]It is important to note that the motivation and full derivation (e.g., why they put a weight of 0.15 on the test coverage, etc.) of the metrics is beyond the scope of this paper. We refer interested readers to the *npms* documentation for more details [11]. To make our paper self-sufficient, we include how the metrics are calculated here.

Table 3: Mann-Whitney Test (p-value) and Cliff's Delta (d) for Trivial vs. Non Trivial Packages

Metrics	p**-value**	d
Tests	2.2e-16	-0.119 (small)
Community interest	2.2e-16	-0.269 (small)
Downloads count	2.2e-16	-0.245 (small)

As an initial step, we calculate the number of trivial packages that have a $Tests$ value greater than zero, which means trivial packages that have some of tests. We find that only 45.2% of the trivial packages have tests, i.e., a $Tests$ value > 0. In addition, we compare the values of the Tests, Community interest and Download count for Trivial and non-Trivial packages. Our focus is on the values of the aforementioned metric values for trivial packages, however, we also present the results for non-trivial packages to put our results in context.

Figure 3 shows the bean-plots for the Tests, Community interest and Download count. The figures show that in all cases trivial packages have, on median, a smaller Tests value, Community interest value and Download count compared to non-trivial packages. That said, we observe from Figure 3 a) that the distribution of the Tests metric is similar for both, trivial and non-trivial packages. Most packages have a Tests value of zero, then there are small pockets of packages that have values of aprox. 0.25, 0.6, 0.8 and 1.0. In the case of the Community interest and Download count metrics, once again, we see similar distributions, although clearly the median values are lower for trivial packages.

To examine whether the difference in metric values between trivial and non-trivial packages is statistically significant, we performed a Mann-Whitney test to compare the two distributions and determine if the difference is statistically significant, with a p-value < 0.05. We also use Cliff's Delta (d), which is a non-parametric effect size measure to interpret the effect size between trivial and non-trivial packages. As suggested in [23], we interpret the effect size value to be small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Table 3 shows the p-values and effect size values. We observe that in all cases the differences are statistically significant, however, the effect size is small. The results show that although the majority of trivial packages do not have tests written for them, and have statistically lower Tests, Community interest, and Download count values, their effect size is smaller than non-trivial packages.

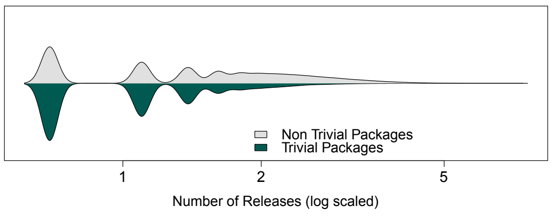


Figure 4: Number of Releases for Trivial Packages Compared to Non-trivial Packages.

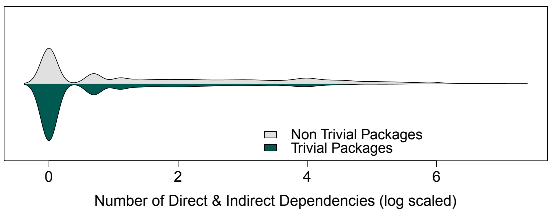


Figure 5: Distribution of Direct & Indirect Dependencies for Trivial and Non-trivial Packages (p-value < 2.2e-16 & Cliff's Delta (d) -0.279 (small)).

Contrary to developers' perception, only 45.2% of trivial packages actually have tests. Albeit, trivial packages have lower Tests, Community interest and Download count values, the values of the metrics do not seem to have a large difference compared to non-trivial packages, i.e., trivial packages are similar to non-trivial packages in terms of how well they are tested.

6.2 Examining the 'Dependency Overhead' Perception

As discussed in Section 5, the top cited drawback of using trivial packages is the fact that developers need to take on and maintain extra dependencies, i.e, dependency overhead. Examining the impact of dependencies is a complex and well-studied issue (e.g,. [1, 12, 15]) that can be examined in a multitude of ways. We choose to examine the issue from both, the application and the package perspectives.
Applications: When compared to coding trivial tasks themselves, using a trivial package imposes extra dependencies. One of the most problematic aspects of managing dependencies for applications is when these dependencies update, causing a potential to break their application. Therefore, as a first step, we examined the number of releases for trivial and non-trivial packages. The intuition here is that developers need to put in extra effort to assure the proper integration of new releases. Figure 4 shows that trivial packages have less releases than non-trivial packages (median is 2 for trivial and 3 for non-trivial packages), hence trivial packages do not require more effort than non-trivial packages. The fact that the trivial packages are updated less frequently may be attributed to the fact that trivial packages 'perform less functionality', hence they need to be updated less frequently.

Table 4: Percentage of Packages vs. the Number of Dependencies Used.

npm Packages	# Dependencies (Direct & Indirect)			
	zero	1-10	11-20	>20
Trivial	56.3%	27.9%	4.3%	11.5%
Non Trivial	34.8%	30.6%	7.3%	27.3%

Next, we examined how developers choose to deal with the updates of trivial packages. One way that application developers reduce the risk of a package impacting their application is to 'version lock' the package. Version locking a dependency/package means that it is not updated automatically, and that only the specific version mentioned in the packages.json file is used. As stated in a few responses from our survey, e.g., P8: *"[...] Also, people who don't lock down their versions are in for some pain."*. There are different types of version locks, i.e., only updating major releases, updating patches only, updating minor releases or no lock at all, which means the package automatically updates. The version locks are specified in the packages.json file next to every package name. We examined the frequency at which trivial and non-trivial packages are locked. We find that on average, trivial packages are locked 14.9% of the time, whereas non-trivial packages are locked 11.7% of the time. However, the Wilcox test shows that the difference is not statistically significant, p-value > 0.05. Hence, we cannot say that developers version lock trivial packages more.
Packages: At the package level, we investigate the direct and indirect dependencies of trivial packages. In particular, we would like to determine if the trivial packages have their own dependencies, which makes the dependency chain even more complex. For each trivial and non-trivial package, we install it and then count the actual number of (direct and indirect) dependencies that the package requires. Doing so, allows us to know the true (direct and indirect) dependencies that each package requires. Note that simply looking into the `.json` file and the `require` statements will provide the direct dependencies, but not the indirect dependencies.

Figure 5 shows the distribution of dependencies for trivial and non-trivial packages. Since most trivial packages have no dependencies, the median is 0. Therefore, we bin the trivial packages based on the number of their dependencies and calculate the percentage of packages in each bin. Table 4 shows the percentage of packages and their respective number of dependencies. We observe that the majority of trivial packages (56.3%) have zero dependencies, 27.9% have between 1-10 dependencies, 4.3% have between 11-20 dependencies and 11.5% have more than 20 dependencies. The table shows that some of the trivial packages have many dependencies, which indicates that indeed, trivial packages can introduce significant dependency overhead.

Trivial packages have fewer releases and developers are less likely to be version locked than non-trivial packages. That said, developers should be careful when using trivial packages, since in some cases, trivial packages can have numerous dependencies. In fact, we find that 43.7% of trivial packages have at least one dependency and 11.5% of trivial packages have more than 20 dependencies.

7 RELEVANCE AND IMPLICATIONS

A common question that is asked in empirical studies is - *so what? what are the implications of your findings? why would practitioners care about your findings?* We discuss the issue of relevance of our study to the developer community, based on the responses of our survey and highlight some of the implications of our study.

7.1 Relevance: Do Practitioners care?

At the start of the study, we were not sure how practically relevant our study of trivial packages is. However, we were surprised by the interest of developers in our study. In fact, one of the developers (P39) explicitly mentioned the lack of research on this topic, stating *"There has not been enough research on this, but I've been taking note of people's proposed "quick and simple" code to handle the functionality of trivial packages, and it's surprised me to see the high percentage of times the proposed code is buggy or incomplete."*

Moreover, when we conducted our study, we asked respondents if they would like to know the outcome of our study and if so, they provide us with an email address. Of the 88 respondents, 66 (aprox. 74%) of them provided their email for us to provide them with the outcomes of our study. Some of these respondents hold very high level leadership roles in *npm*. To us this is an indicator that our study and its outcomes are of high relevance to the *npm* and Node.js development community.

7.2 Implications of Our Study

Our study has a number of implications on both, software engineering research and practice.

Implications for Future Research: Our study mostly focused on determining the prevalence, reasons for and drawbacks of using trivial packages. Based on our findings, we find a number of implications/motivations for future work. First, our survey respondents indicated that the choice to use trivial packages is not black or white. In many cases, it depends on the team and the project. For example, one survey respondent stated that on his team, less experienced developers are more likely to use trivial packages, whereas the more experienced developers would rather write their own code for trivial tasks. The issue here is that the experienced developers are more likely to trust their own code, while the less experienced are more likely to trust an external package. Another aspect is the maturity of the project. As some of the survey respondents pointed out, they are much more likely use trivial packages early on in the project, so they do not waste time on trivial tasks and focus on the more fundamental tasks of their project. However, once their project matures, they start to look for ways to reduce dependencies since they pose potential points of failure for their project. Hence, our study motivates future work to examine the relationship between team experience and project maturity and the use of trivial packages.

Second, survey respondents also pointed out that using trivial packages is seen favourably compared to using code from Q&A sites such as StackOverflow or Reddit. When compared to using code on StackOverflow, where the developer does not know who posted the code, who else uses it or whether the code may have tests or not, using a trivial package that is on *npm* is a much better option. In this case, using trivial packages is not seen as the *best* choice, but it is certainly a better choice. Although there have been many studies that examined how developers use Q&A sites such as StackOverflow, we are not aware of any studies that compare code reuse from Q&A sites and trivial packages. Our findings motivate the need for such a study.

Practical Implications: A direct implication of our findings is that trivial packages are commonly used by others, perhaps indicating that developers do not view their use as bad practice. Moreover, developers should not assume that all trivial packages are well implemented and tested, since our findings show otherwise. *npm* developers need to expect more trivial packages to be submitted, making the task of finding the most relevant package even harder. Hence, the issue of how to manage and help developers find the best packages needs to be addressed. To some extent, *npms* has been recently adopted by *npm* to specifically address the aforementioned issue. Developers highlighted that the lack of a decent core or standard JavaScript library causes them to resort to trivial packages. Often, they do not want to install large frameworks just to leverage small parts of the framework, hence they resort to using trivial packages. Therefore, there is a need by the Node.js community to create a standard JavaScript API or library in order to reduce the dependence on trivial packages. However, the issue of creating such a standard JavaScript library is under much debate [20].

8 RELATED WORK

Studies of Code Reuse. Prior research on code reuse has been shown its many benefits, which include improving quality, development speed, and reducing development and maintenance costs [3, 32, 36, 37]. For example, Sojer and Henkel [43] surveyed 686 open source developers to investigate how they reuse code. Their findings show that more experienced developers reuse source code and 30% of the functionality of open source software (OSS) projects reuse existing components. Developers also reveal that they see code reuse as a quick way to start new projects. Similarly, Haefliger *et al.* [24] conducted a study to empirically investigate the reuse in open source software, and the development practices of developers in OSS. They triangulated three sources of data (developer interviews, code inspections and mailing list data) of six OSS projects. Their results showed that developers used tools and relied on standards when reusing components. Mockus [36] conducted an empirical study to identify large-scale reuse of open source libraries. Their study shows that more than 50% of source files include code from other OSS libraries. On the other hand, the practice of reusing source code has some challenging drawbacks including the effort and resource required to integrate reused code [16]. Furthermore, a bug in the reused component could propagate to the target system [17]. While our study corroborates some of these findings, the main goal is to define and empirically investigate the phenomenon of reusing trivial packages, in particular in Node.js applications.

Studies of Other Ecosystems. In recent years, analyzing the characteristics of ecosystems in software engineering has gained momentum [4, 5, 15, 34]. For example, in a recent study, Bogart *et al.* [6, 7] empirically studied three ecosystems, including *npm*, and found that developers struggle with changing versions as they might break dependent code. Witter *et al.* [46] investigated the evolution of the *npm* ecosystem in an extensive study that covers the dependence between *npm* packages, download metrics and the usage of *npm* packages in real applications. One of their main findings is that *npm* packages and updates of these packages is steadily growing. Also, more than 80% of packages have at least one direct dependency package.

Other studies examined the size characteristics of packages in an ecosystem. German *et al.* [21] studied the evolution of the statistical computing project GNU R, with the aim of analyzing the differences between code characteristics of core and user-contributed packages. They found that user-contributed packages are growing faster than core packages. Additionally, they reported that user-contributed packages are typically smaller than core packages in the R ecosystem. Kabbedijk and Jansen [30] analyzed the Ruby ecosystem and found that many small and large projects are interconnected.

In many ways, our study complements the previous work since, instead of focusing on all packages in an ecosystem, we specifically focus on trivial packages. Moreover, we examine the reasons developers use trivial package and what they view as their drawbacks.

We study the reuse of trivial packages, which is a subset of general code reuse. Hence, we do expect there to be some overlap with prior work. Like many empirical studies, we confirm some of the prior findings, which is a contribution on its own. Moreover, our paper adds to the prior findings through, for example, our validation of the developers' assumptions. Lastly, we do believe our study fills a real gap since 74% of the participants said they wanted to know our study outcomes.

9 THREATS TO VALIDITY

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. To define trivial packages, we surveyed 12 JavaScript developers who are mostly graduate student with some professional experience. However, we find that there was a clear vote for what is considered a trivial package. Also, although our data suggested that packages with ≤ 35 LOC and a complexity ≤ 10 are trivial packages, we believe that other definitions are possible for trivial packages. That said, of the 88 survey participants that we emailed about using trivial packages, only 1 mentioned that the flagged package is not a trivial package (even though it fit our criteria). To us, this is a confirmation that our definition applies in the vast majority of the cases, although clearly it is not perfect.

We use the LOC and complexity of the code to determine trivial packages. In some cases, these may not be the only measures that need to be considered to determine a trivial packages. For example, some of the trivial packages have their own dependencies, which may need to be taken into consideration. However, our experience tells us that most developers only look at the package itself and not its dependencies when determining if it is trivial or not. That said, it would be interesting to replicate this questionnaire with another set of participants to confirm or enhance our definition of a trivial Node.js package.

Our list of reasons for and drawbacks of using trivial packages are based on a survey of 88 Node.js developers. Although this is a large number of developers, our results may not hold for all Node.js developers. A different sample of developers may result in a different list or ranking of advantages and disadvantages. To mitigate the risk due to this sampling, we contacted developers from different applications and as our responses show, most are experienced developers. Also, there is potential that our survey questions may have influenced the replies from the respondents. However, to minimize such influence, we made sure to ask for free-form responses (to minimize any bias) and we publicly share our survey and all of our anonymized survey responses.

We used *npms* to measure various quantitative metrics related to testing, community interest and download counts. Our measurements are only as accurate as *npms*, however, given that it is the main search tool for *npm*, we are confident in the the *npms* metrics.

We do not distinguish between the domain of the *npm* packages, which may impact the findings. However, to help mitigate any bias we analyzed more than 230,000 *npm* packages that cover a wide range of domains.

We removed test code from our dataset to ensure that our analysis only considers JavaScript source code. We identified test code by searching for the term 'test' (and its variants) in the file names and file paths. Even though this technique is widely accepted in the literature [22, 44, 48], to confirm whether our technique is correct, i.e., files that have the term 'test' in their names and paths actually contain test code, we took a statistically significant sample of the packages to achieve a 95% confidence level and a 5% confidence interval and examined them manually.

External validity considers the generalization of our findings. All of our findings were derived from open source Node.js applications and *npm* packages, hence, our findings may not generalize to other platforms or ecosystems. That said, historical evidence shows that examples of individual cases contributed significantly in areas such as physics, economics, social sciences and even software engineering [19]. We believe that strong empirical evidence is built from both, studies on individual cases and studies on large samples.

10 CONCLUSION

The use of trivial packages is an increasingly popular trend in software development. Like any development practice, it has its proponents and opponents. The goal of our study is to examine the prevalence, reasons and drawbacks of using trivial packages. Our findings indicate that trivial packages are commonly and widely used in Node.js applications. We also find that the majority of developers do not oppose the use of trivial packages and the main reasons developers use trivial packages is due to the fact that they are considered to be well implemented and tested. However, they do cite the fact that the additional dependencies' overhead as a drawback of using these trivial packages. That said, our empirical study showed considering trivial packages to be well tested is a misconception since more than half of the trivial package we studied do not even have tests written, however, these trivial packages seem to be 'deployment tested' and have similar Tests, Community interest and Download count values as non-trivial packages. In addition, we find that some of the trivial packages have their own dependencies and, in our studied dataset, 11.5% of the trivial packages have more than 20 dependencies. Hence, developers should be careful about which trivial packages they use.

11 ACKNOWLEDGMENTS

The authors are grateful to the many survey respondents who dedicated their valuable time to respond to our surveys.

REFERENCES

[1] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. 2009. Strong Dependencies Between Software Components. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*. IEEE Computer Society, 89–99.

[2] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. 2017. On Code Reuse from StackOverflow : An exploratory study on Android apps. *Information and Software Technology* 88, C (2017), 148–158.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. 1996. How Reuse Influences Productivity in Object-oriented Systems. *Commun. ACM* 39, 10 (October 1996), 104–116.

[4] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, 280–289.

[5] Remco Bloemen, Chintan Amrit, Stefan Kuhlmann, and Gonzalo Ordóñez Matamoros. 2014. Gentoo Package Dependencies over Time. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. ACM, 404–407.

[6] Christopher Bogart, Christian Kastner, and James Herbsleb. 2015. When It Breaks, It Breaks: How Ecosystem Developers Reason About the Stability of Dependencies. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW '15)*. IEEE Computer Society, 86–89.

[7] 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 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 109–120.

[8] Stephan Bonnemann. 2015. Dependency Hell Just Froze Over. https://speakerdeck.com/boennemann/dependency-hell-just-froze-over. (September 2015). (accessed on 08/10/2016).

[9] Bower. 2012. Bower a package manager for the web. https://bower.io/. (2012). (accessed on 08/23/2016).

[10] J. Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20 (1960), 37–46.

[11] Andre Cruz and Andre Duarte. 2017. npms. https://npms.io/. (01 2017). (accessed on 02/20/2017).

[12] 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, 241–250.

[13] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the Topology of Package Dependency Networks: A Comparison of Three Programming Language Ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops (ECSAW '16)*. ACM, Article 21, 4 pages.

[14] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems. In *Proccedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE.

[15] Alexandre Decan, Tom Mens, Philippe Grosjean, and others. 2016. When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1. IEEE, 493–504.

[16] Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. 2011. Supporting software evolution in component-based FOSS systems. *Science of Computer Programming* 76, 12 (2011), 1144–1160.

[17] Mehdi Dogguy, Stephane Glondu, Sylvain Le Gall, and Stefano Zacchiroli. 2011. Enforcing Type-Safe Linking using Inter-Package Relationships. *Studia Informatica Universalis.* 9, 1 (2011), 129–157.

[18] J. L. Fleiss and J. Cohen. 1973. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and Psychological Measurement* 33 (1973), 613–619.

[19] Bent Flyvbjerg. 2006. Five misunderstandings about case-study research. *Qualitative Inquiry* 12, 2 (2006), 219–245.

[20] Thomas Fuchs. 2016. What if we had a great standard library in JavaScript? fi?! Medium. https://medium.com/@thomasfuchs/what-if-we-had-a-great-standard-library-in-javascript-52692342ee3f.pw7d4cq8j. (Mar 2016). (accessed on 02/24/2017).

[21] D German, B Adams, and AE Hassan. 2013. Programming language ecosystems: the evolution of r. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE, 243–252.

[22] Georgios Gousios and Andy Zaidman. 2014. A Dataset for Pull-based Development Research. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. ACM, 368–371.

[23] Robert J Grissom and John J Kim. 2005. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers.

[24] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. 2008. Code reuse in open source software. *Management Science* 54, 1 (2008), 180–193.

[25] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 144–156.

[26] David Haney. 2016. NPM & left-pad: Have We Forgotten How To Program? http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/. (March 2016). (accessed on 08/10/2016).

[27] Rich Harris. 2015. Small modules: itfis not quite that simple. https://medium.com/@Rich_Harris/small-modules-it-s-not-quite-that-simple-3ca532d65de4. (Jul 2015). (accessed on 08/24/2016).

[28] Hemanth.HM. 2015. One-line node modules -Issue#10- sindresorhus/ama. https://github.com/sindresorhus/ama/issues/10. (2015). (accessed on 08/10/2016).

[29] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. 2012. Where Does This Code Come from and Where Does It Go? - Integrated Code History Tracker for Open Source Systems -. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 331–341.

[30] Jaap Kabbedijk and Slinger Jansen. 2011. Steering insight: An exploration of the ruby software ecosystem. In *Proceedings of the Second International Conference of Software Business (ICSOB '11)*. Springer, 44–55.

[31] 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 '14)*. ACM, 92–101.

[32] Wayne C. Lim. 1994. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software* 11, 5 (1994), 23–30.

[33] Fiona Macdonald. 2016. A programmer almost broke the Internet last week by deleting 11 lines of code. &+#http://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code. (March 2016). (accessed on 08/24/2016).

[34] Konstantinos Manikas. 2016. Revisiting software ecosystems research: a longitudinal literature study. *Journal of Systems and Software* 117 (2016), 84–103.

[35] Stephen McCamant and Michael D. Ernst. 2003. Predicting Problems Caused by Component Upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '03)*. ACM, 287–296.

[36] Audris Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS '07)*. IEEE Computer Society, 7–.

[37] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. 2004. An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, 282–292.

[38] npm. 2016. Most depended-upon packages. http://www.npmjs.com/browse/depended. (August 2016). (accessed on 08/10/2016).

[39] npm. 2016. What is npm? — Node Package Managment Documentation. https://docs.npmjs.com/getting-started/what-is-npm. (July 2016). (accessed on 08/14/2016).

[40] The npm Blog. 2016. The npm Blog changes to npm's unpublish policy. http://blog.npmjs.org/post/141905368000/changes-to--unpublish-policy. (March 2016). (accessed on 08/11/2016).

[41] Heikki Orsila, Jaco Geldenhuys, Anna Ruokonen, and Imed Hammouda. 2008. Update propagation practices in highly reusable open source components. In *Proceedings of the 4th IFIP WG 2.13 International Conference on Open Source Systems (OSS '08)*. 159–170.

[42] Janice Singer, Susan E Sim, and Timothy C Lethbridge. 2008. Software engineering data collection for field studies. In *Guide to Advanced Empirical Software Engineering*. Springer London, 9–34.

[43] Manuel Sojer and Joachim Henkel. 2010. Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. *Journal of the Association for Information Systems* 11, 12 (2010), 868–901.

[44] 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 '14)*. ACM, 356–366.

[45] Chris Williams. 2016. How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos. (March 2016). (accessed on 08/24/2016).

[46] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, 351–361.

[47] Dan Zambonini. 2011. Testing and deployment. In *A Practical Guide to Web App Success*, Owen Gregory (Ed.). Five Simple Steps, Chapter 20. (accessed on 02/23/2017).

[48] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2014. Patterns of Folder Use and Project Popularity: A Case Study of Github Repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, Article 30, 4 pages.