Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09792-9

On the impact of using trivial packages: an empirical case study on *npm* and *PyPI*

Rabe Abdalkareem[1] · Vinicius Oda[1] · Suhaib Mujahid[1] · Emad Shihab[1]

Abstract

Code reuse has traditionally been encouraged since it enables one to avoid re-inventing the wheel. Due to the *npm* left-pad package incident where a trivial package led to the breakdown of some of the most popular web applications such as Facebook and Netflix, some questioned such reuse. Reuse of trivial packages is particularly prevalent in platforms such as *npm*. To date, there is no study that examines the reason why developers reuse trivial packages other than in *npm*. Therefore, in this paper, we study two large platforms *npm* and *PyPI*. We mine more than 500,000 *npm* packages and 38,000 JavaScript applications and more than 63,000 *PyPI* packages and 14,000 Python applications to study the prevalence of trivial packages. We found that trivial packages are common, making up between 16.0% to 10.5% of the studied platforms. We performed surveys with 125 developers who use trivial packages to understand the reasons and drawbacks of their use. Our surveys 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 validate the most cited reason and drawback. We find that contrary to developers' beliefs only around 28% of *npm* and 49% *PyPI* trivial packages 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 18.4% and 2.9% of the studied trivial packages have more than 20 dependencies in *npm* and *PyPI*, respectively.

Keywords Trivial packages · JavaScript · Node.js · Python · *npm* · *PyPI* · Code reuse · Empirical studies

1 Introduction

Code reuse, in the form of combining related functionalities in packages, has been encouraged due to the fact that it can reduce the time-to-market, improve software quality and

Communicated by: Arie van Deursen

✉ Rabe Abdalkareem
rab_abdu@encs.concordia.ca

Extended author information available on the last page of the article.

boost overall productivity (Basili et al. 1996; Lim 1994; Mohagheghi et al. 2004). Therefore, it is no surprise that platforms such as Node.js encourage reuse and attempt to facilitate code sharing, often delivered as packages or modules[1] that are available on package management platforms, such as the Node Package Manager (*npm*) and Python Package Index (*PyPI*) (npm 2016; Bogart et al. 2016).

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 (McCamant and Ernst 2003; Orsila et al. 2008; Inoue et al. 2012; Abdalkareem et al. 2017a). For example, an incident of code reuse of a JavaScript 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' (Macdonald 2016; Williams 2016). That incident lead to many heated discussions about code reuse, sparked by David Haney's blog post: *"Have We Forgotten How to Program?"* (Haney 2016).

While the real reason for the left-pad incident was that *npm* allowed authors to unpublish packages (a problem which has been resolved (npm Blog 2016)), it raised awareness of the broader issue of taking on dependencies for trivial tasks that can be easily implemented (Haney 2016). In our previous work (Abdalkareem et al. 2017), we defined and examined trivial packages in *npm*, and discovered a number of relevant findings:

- Trivial JavaScript packages tend to be small in size and less complex.
- Trivial packages are prevalent, making up approximately 16.8% of all the packages on *npm*.
- JavaScript developers generally use trivial packages since they believe that trivial packages provide them with well tested and implemented code, however, they are concerned about the management of extra dependencies.

In addition, we found that in some cases, these trivial JavaScript packages can have their own dependencies, imposing significant overhead.

However, one major limitation of the original work was its deep focus on JavaScript and *npm* in particular (Abdalkareem et al. 2017). For example, questions about the existence of trivial packages (and how they are defined) in other package management platforms remain. Also, whether the perceived advantages (e.g., that trivial packages are well tested) and disadvantages (e.g., management of additional dependencies) of using trivial packages generalized beyond JavaScript developers remain unanswered.

Hence, this paper has extended our previous work (Abdalkareem et al. 2017) to strengthen the empirical evidence on the use of trivial packages by replicating and extending our study on the Python Package Index (*PyPI*). We chose to examine the *PyPI* package management platform since 1) Python is one of the most popular general purpose programming languages, 2) Python has only one main well-established package platform, *PyPI*, and 3) *PyPI* is a mature package management platform that has been in existence for more than twelve years. Our extended study provides the following key additions:

- We extended our study of the *npm* package management platform and increased the *npm* dataset from 231,092 to 501,001 packages.
- We provide a definition of *PyPI* trivial packages and examine the prevalence of trivial packages in the Python ecosystem.

[1] In this paper, we use the term package to refer to a software library that is published on the studied package management platforms.

- We surveyed 37 Python developers to investigate the reasons for and drawback of using trivial packages in the *PyPI* package management platform.
- We examine the top main reasons of and drawbacks of using *PyPI* trivial packages based on the developers survey.

Altogether, our study involves more than 500,000 *npm* packages and 38,000 JavaScript applications and 63,000 *PyPI* packages and 14,000 Python applications. The study also contains survey results from 125 JavaScript and Python developers. Our findings indicate that:

The definition of trivial packages is the same in JavaScript and Python The developers from the two different package management platforms tended to have the same definition of trivial packages. While we found in the original paper (Abdalkareem et al. 2017) that *npm* trivial packages are packages that have ≤ 35 LOC and a McCabe's cyclomatic complexity ≤ 10, we also found that *PyPI* trivial packages have the same definition.

Trivial packages are common and popular in both, *npm* and *PyPI* management platforms Of the 501,001 *npm* and 63,912 *PyPI* packages in our dataset, 16.0% and 10.6% of them are trivial packages. Moreover, of the 38,807 JavaScript and 14,717 Python applications on GitHub, 26.1% and 6.9% of them directly depend on one or more trivial packages.

JavaScript and Python developers differ in their perception of trivial packages Only 23.9% of JavaScript developers considered the use of trivial packages as bad, whereas, 70.3% of Python developers consider the use of trivial package as a bad practice.

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 *npm*, 43.2% of trivial packages have at least one dependency and 18.4% of trivial packages have more than 20 dependencies. In *PyPI*, 36.8% of trivial packages have at least one dependency, and 2.9% have more than 20 dependencies.

To facilitate the replicability of our work, we make our dataset and the anonymized developer responses publicly available (Abdalkareem et al. 2019).

1.1 Paper Organization

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 JavaScript and Python applications. Section 5 presents the results of our developer surveys, presenting the reasons and perceived drawbacks for developers who use trivial packages. Section 6 presents our quantitative 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 Case Studies

In this section, we provide background on the two studied package management platforms, *npm* and *PyPI*. We also provide an overview of the dataset collected and used in the rest of our study.

2.1 Node Package Manager (*npm*)

JavaScript is used to write client and server side applications. The popularity of JavaScript has steadily grown, thanks to popular frameworks such as Node.js and an active developer community (Bogart et al. 2016; Wittern et al. 2016). JavaScript projects can be classified into two main categories: *JavaScript packages* that are used in other applications or *JavaScript applications* that are used as standalone software. The Node Package Manager (*npm*) provides tools to manage JavaScript packages.

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

***npm* Packages:** Since we are interested in examining the impact of 'trivial packages', we mined the latest version of all the JavaScript packages from *npm* as of September 30, 2017. For each package we obtained its source code from the *npm* registry. In total, we mined 549,629 packages.

GitHub JavaScript Applications: We also want to examine the use of the *npm* packages in JavaScript applications. Therefore, we mined all of the JavaScript applications on GitHub. To obtain a list of JavaScript applications, we extracted all the applications identified as JavaScript application from the GHTorrent dataset (Gousios et al. 2014). Then, to ensure that we are indeed only obtaining the JavaScript applications from GitHub, and not *npm* packages, we compare the URL of the GitHub repositories from GHTorrent 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.

Finally, 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 used in prior work by Kalliamvakou et al. (2014). 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 JavaScript applications.

2.2 Python Package Index (*PyPI*)

PyPI is the official package management platform for the Python programming language. Python is one of the most popular programming language today, mainly due to its strong community support and versatility, i.e., Python is used in many different domains from game development to server side applications (Vasilescu et al. 2015; Ray et al. 2014). Once again, we distinguish between *Python packages*, which are used in Python applications and standalone *Python applications*, which typically use Python packages. Similar to the case of JavaScript, we gather two datasets from two sources to perform our study. We obtain Python packages from the *PyPI* registry and applications that use *PyPI* packages from GitHub.

***PyPI* Packages:** We collected the latest versions of the Python packages from *PyPI* in order to determine which packages are 'trivial packages'. *PyPI* contains around 118,324 packages (Libraries.io 2017), as of September 30, 2017. In total, we were able to obtain 116,905 packages from the *PyPI* registry since some packages did not exist anynmore.

GitHub Python Applications: To examine the usage of 'trivial packages' in Python applications, we mined all of the Python applications hosted on GitHub provided by the GHTorrent dataset (Gousios et al. 2014). We followed the same aforementioned process used to gather JavaScript applications, to ensure that we are indeed only obtaining the Python applications from GitHub, and not *PyPI* package repositories. In a nutshell, we compare the URL of the GitHub repositories to the URLs we obtained from *PyPI* for the packages. If a URL from GitHub was also in *PyPI*, we flagged it as being an *PyPI* package and removed it from the application list. In total, we obtained 14,717 Python applications that are hosted on GitHub. In addition, to eliminate dummy or immature Python applications that may exist in GitHub, we performed the filtering steps as we did for the JavaScript application. We choose non-forked Python applications with more than 100 commits and more than 2 developers.

3 Defining Trivial Packages

Although what a trivial package is has been loosely defined in the past (e.g., in blogs (Hemanth 2015; Harris 2015)), we want a more precise and objective way to determine trivial packages. To determine what constitutes a trivial package, we conducted two separate surveys, one for each of the studied package management platforms (*npm* and *PyPI*). We mainly 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 conducted two different surveys since: 1) the two studied package management platforms serve different programming languages, 2) developers from the two package management platforms may have different perspective of what they consider to be 'trivial packages'.

For each package management platform (*npm* and *PyPI*), we devised an online survey that presented the source code of 16 randomly selected packages that range in size between 4 - 250 JavaScript/Python 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 limit the size of the selected packages in the surveys to a maximum of 250 JavaScript/Python 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 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 JavaScript 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

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

Fig. 1 Package is-Positive on *npm*

background, 2) questions about the classification of the provided packages, and 3) questions about what indicators the participant would use to determine a trivial package. For the *npm* survey, we sent the survey to 22 developers and colleagues that were familiar with JavaScript development and received a total of 12 responses. We also sent the *PyPI* survey to 18 developers and colleagues that were familiar with Python development and received a total of 13 responses. It is important to note that we sent the two surveys to different groups of developers, to make sure that the participants in one survey are not biased through their experience of participating in the other (i.e., first) survey.

Participants' Background and Experience: The first four columns of Table 1 show the background of participants in the *npm* survey. 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.

The last four columns of Table 1 show the background of participants in the *PyPI* survey. Of the 13 participants in this survey, 9 identified themselves as graduate students and 4 as professional developers working in industry; 7 participants had more than 5 years of Python development experience, 2 respondents had between 3 to 5 years, 3 others had 2 to 3 years of experience, and finally one person had less than 1 year of Python practice. We were happy to have the majority of our respondents be well-experienced with Python.

Result: We asked participants of the two surveys 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 in the *JavaScript* survey, 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. Also, 3 (20%) mentioned that they used code comments and other indicators (e.g., functionality) to indicate if a package is trivial or not. The results of the *Python* survey reveal that 9 (69%) of the developers use size of the code and 9 (69%) of them use complexity of the code as the main indicators to determine trivial packages. Also, 7 (54%) of the participants stated that they use source code comments to determine trivial Python packages and 3 (23%) of the participants mentioned some other indicators that they can use to identify a trivial package. For example one participant related a trivial Python package as *"If it's only one function"*.

Table 1 Background of participants in the two surveys to determine trivial packages

npm				PyPI			
Experience in JavaScript	#	Developers' position	#	Experience in python	#	Developers' position	#
<1	2	Undergrad Student	2	<1	1	Undergrad Student	0
2 – 3	3	Graduate Student	8	2 – 3	3	Graduate Student	9
3 – 5	1	Professional Developer	2	3 – 5	2	Professional Developer	4
>5	6	–	–	>5	7	–	–
Total	12	Total	12	Total	13	Total	13

Since it is clear that size and complexity are the most common indicators of trivial packages and they are a universal measure that can be measured for both, JavaScript and Python, we use these two measures to determine trivial packages. It should be mentioned that participants could provide more than one indicator, hence the percentages above sum to more than 100%.

Next, we analyze all of the packages that were marked as trivial from the two surveys. Our main goal of this analysis is to find which values of the size and complexity metrics are indicative of trivial packages.

***npm* Survey Responses:** 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 with 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 to not bias them towards any specific metrics.

***PyPI* Survey Responses:** we received 89 votes for the 16 packages. Similar to the case of *npm*, we ranked the packages in ascending order, based on their size, and tallied the votes for the most voted packages. We find that 76.4% of the votes consider packages that are equal or 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 79.8% of the votes marked packages that have a total complexity value of 10 or lower to be trivial Python package. Similar to *npm*, we also did not provide any metric values for the packages to avoid bias.

Based on the aforementioned findings, we used the two indicators JavaScript/Python LOC $\leq$ 35 and complexity $\leq$ 10 to determine trivial packages in our dataset. Hence, we define trivial JavaScript/Python packages as $\{X_{LOC} \leq 35 \cap X_{Complexity} \leq 10\}$, where X_{LOC} represents the JavaScript/Python 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.

> ***Our analysis shows that trivial packages for the two studied package management platforms (npm and PyPI) have similar definition. As our two surveys show, size and complexity are commonly used measures to determine if a package is trivial for both JavaScript and Python programming languages. Based on our analysis, JavaScript and Python packages that have $\leq$ 35 LOC and a McCabe's cyclomatic complexity $\leq$ 10 are considered to be trivial packages.***

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 package management platforms (*npm* and *PyPI*) perspective, where we are interested in knowing how many of the packages on these two

package management platforms are trivial. The second aspect considers the use of trivial packages in JavaScript and Python applications.

To identify trivial packages in our two datasets, we calculate the LOC and complexity of all the *npm* and *PyPI* packages. For the LOC, we calculate the number of lines of source code after removing white space and source code comments. As for the complexity, we use McCabe's complexity since it is widely used in industry and academia (Ebert and Cain 2016). 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 (Gousios et al. 2014; Tsay et al. 2014; Zhu et al. 2014), we look for the term "test" (and its variants such as 'tests' and/or 'TEST_code') in the file names and file paths. To calculate the LOC and the complexity of every package in our datasets, we use the Understand tool by SciTools (https://scitools.com/). Understand is a source code analysis tool that provides various code metrics and has been extensively used in other work (e.g., Rahman et al. 2019; Castelluccio et al. 2019).

4.1 How Many of *npm*'s & *PyPI*'s Packages are Trivial?

***npm*:** 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 549,629 *npm* packages. For each package, we calculated the number of JavaScript code lines and removed packages that had zero LOC, which removed 48,628 packages. We eliminated *npm* packages that have zero LOC since they present dummy or empty packages that developers publish for different reasons such as reserve a unique package name. This left us with a final number of 501,001 packages.

Out of the 501,001 *npm* packages we mined, 80,232 (16.0%) 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 published over time before the growth of trivial packages became stable around the beginning of 2015. Overall, approximately 14.0% 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 (npm Blog 2016).

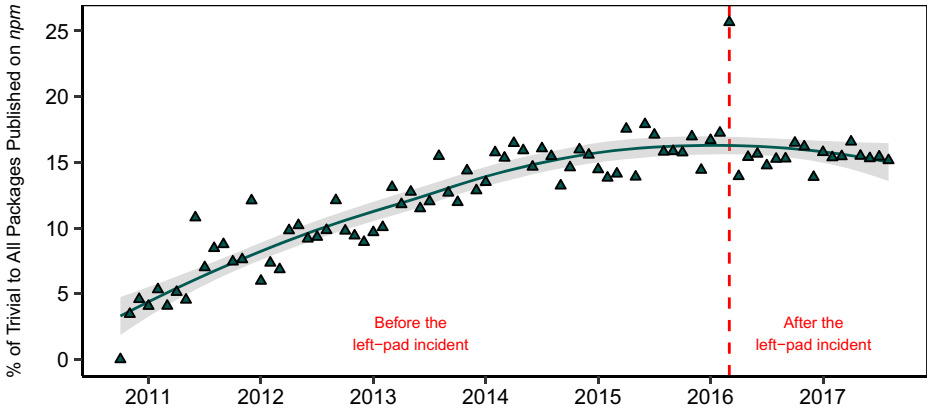


Fig. 2 Percentage of published trivial packages on *npm*. The dashed vertical line represents the date when the left-pad incident happened in *npm* package management platform

In addition, to see the effect of the left-pad incident on the number of published trivial packages, we investigate the number of published trivial *npm* packages before and after the left-pad incident. Out of 216,309 *npm* packages that published before the left-pad incident, we found 34,750 (16.1%) are trivial packages. As after the left-pad incident, out of the 284,692 that are published, we found 45,482 (16.0%) are trivial packages.

***PyPI*:** For the *PyPI* dataset, we are also interested in discerning the trivial packages from the others in terms of LOC and complexity. For such, we mined the 116,905 available packages on the *PyPI* platform. We got all the 116,905 packages from *PyPI* register. However, a package on *PyPI* could be released/distributed in different formats and we were not able to process them. We found that 42,242 of *PyPI* packages are platform exclusive (e.g., windows .exe or mac .dmg) or are corrupted compressed .gz files that we could not analyzed. This process left us with 74,663 *PyPI* packages for which we measure their LOC and complexity. We then remove packages that had zero LOC, which removed another 10,751 packages. We remove packages that had zero LOC since we do not want to count empty packages that exist on *PyPI* for various reasons such as learning to publish packages on *PyPI*.

Our analysis reveals that out of the 63,912 *PyPI* packages we analyzed, 6,759 (10.6%) packages are trivial packages in the *PyPI* package management platform. We again examined the growth of trivial packages in *PyPI*. Figure 3 shows the percentage of trivial to all packages published on *PyPI* per month for the time period between 2011 and 2017. We see there is a slight increase in the trend of publishing trivial packages on the *PyPI* platform and that trend starts to decrease in late 2013. We also found that approximately 11% of the packages added every month are trivial packages.

We also looked at the percentage of trivial to all packages publish before and after the left-pad incident. We found that out of 33,335 *PyPI* package published prior to the left-pad incident, 3,717 (11.2%) of them are trivial packages while 3,042 (10.0%) of all packages published after the left-pad incident are trivial.

Trivial packages make up 16.0% of the studied npm packages and 10.6% of the PyPI packages.

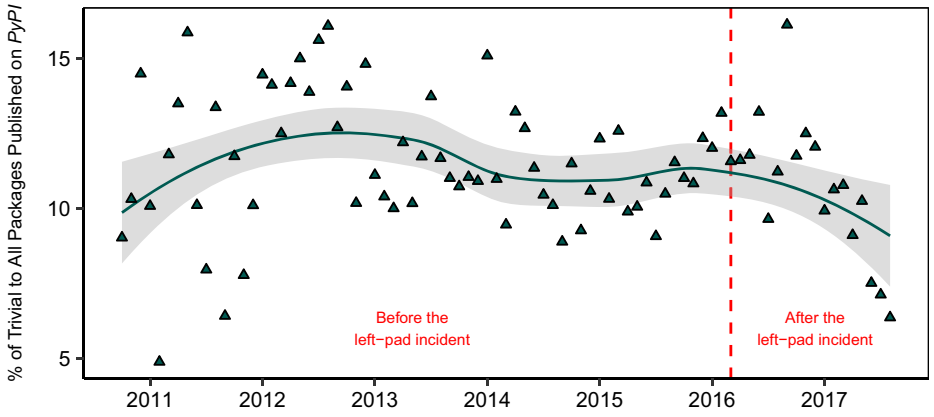


Fig. 3 Percentage of Published Trivial Packages on *PyPI*. The dashed vertical line represents the date when the left-pad incident happened in *npm* package management platform

4.2 How Many Applications Depend on Trivial Packages?

JavaScript Applications: Just because trivial packages exist on *npm*, it does not mean that they are actually being used. 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 required 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 JavaScript packages and other package managers only manage a subset of packages (e.g., Bower (2012) only manages front-end/client-side frameworks, libraries and modules). We find that of the 38,807 applications in our dataset, 10,139 (26.1%) directly depend on at least one trivial package.

Python Applications: Similar to the case of JavaScript, we also analyzed the Python applications that depend on trivial packages. In contrast to JavaScript's availability of a 'packages.json' file, analyzing Python applications presents some challenges to fully identify a given script's dependency set for the reasons described previously on Section 4.1. We statically parse the source code after relevant "import" like clauses, along with other statements that allow for verifying that the packages are effectively being put in use (i.e., the package is both supposed to be installed and its functions/definitions are indeed being called, rather than merely being just imported and not used). To facilitate this analysis, we use the popular `snakefood` (http://furius.ca/snakefood/) tool. The tool generates dependency graphs from Python code through parsing the Abstract Syntax Tree of the Python files. Our analysis showed that out of the 14,717 examined Python applications, 1,024 (6.9%) were found to depend on one or more trivial *PyPI* package.

> ***Of the 38,807 JavaScript applications in our dataset, 10.9% of them depend on at least one trivial package. Out of the 14,717 examined Python applications, 1,024 were found to depend on at least one trivial PyPI package (6.95% of the selected GitHub applications)***

5 Survey Results

We surveyed developers to understand the reasons for and the drawbacks of using trivial packages. We used 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 JavaScript and Python applications. Doing so helped us identify the name and email of the developer who introduced the trivial package dependency.

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

Survey Participants: To mitigate the possibility of introducing misunderstood or misleading questions, we initially sent the survey to two developers and incorporated their minor suggestions to improve the survey. For *npm* participants, we sent the survey to 1,055 JavaScript 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. We also sent the survey to all Python developers after filtering out the invalid and duplicated developers' emails. We successfully sent the survey to 460 Python developers that introduce trivial Python packages from *PyPI* in 1,024 Python applications in our dataset.

We designed the survey using Google Forms. The survey listed the trivial package and the application that we detected the trivial package in. In total, we received 125 developer responses. First, we received 88 responses to our survey from the ***JavaScript*** developers, which translates to a response rate of 8.3%. Our survey response rate is higher than the typical 5% response rate reported in questionnaire-based software engineering surveys (Singer et al. 2008). The left part of Table 2 show the JavaScript experience and the position of the developers. 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. As for the position of the survey respondents, of the 88 respondents, 83 of them identified as developers working either in industry (68) or as full time independent developers (15). The remaining 5 identified as being casual developers (2) or other (3), including one student and two developers working in executive positions at *npm*.

Second, we received 37 survey responses from the ***Python*** developers, yielding a response rate of 8.04%, which is again in accordance with what is supposedly been observed on other studies in the software engineering domain (Singer et al. 2008). The right part of Table 2 shows the Python experience and position of the developers. The vast majority of the respondents (92%) identified themselves to have more than five years of Python development experiences. 3 respondents only identified themselves to have development experience in Pythons range between more than 3 to five years. Regarding the current position of the survey respondents, 27 of the respondents refer themselves as developers working in industry and 4 developers identified themselves as full time independent developers. The reset of the respondents are identified as being a casual developers (1) or other (5) including researchers and students.

Table 2 Development experience and position of survey respondents

npm				PyPI			
Experience in JavaScript	#	Developers' Position	#	Experience in Python	#	Developers' Position	#
1 - 3 years	7	Industrials	68	1 - 3 years	0	Industrials	27
> 3 - 5 years	14	Independent	15	> 3 - 5 years	3	Independent	4
> 5 years	67	Casual	2	> 5 years	34	Casual	1
–	–	Other	3	–	–	Other	5
Total	88	Total	88	Total	37	Total	37

The fact that most of the respondents are experienced JavaScript and Python 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?" Thereason to ask this question so bluntly is that it allows us to gauge, in a very deterministic way, how the 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. Figure 4 shows the distribution of responses from both JavaScript and Python developers. Of the 88 JavaScript 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.

Contrary to the case of JavaScript, 26 (70.3%) of the Python developers who responded to our survey generally consider the use of trivial packages as bad practice. Only 3 (8.1%) of survey participants stated that they do not think that using trivial package is a bad practice. The remaining 8 (21.6%) indicate that it really depends on the circumstances. For example, P-*PyPI* 3 states: *"If the language doesn't provide such common, inherently useful functionality then fixing this oversight by the use of a third-party library is only reasonable. Moreover, little functionality is actually 'trivial'. It may be short to implement but most likely a mistake in it will introduce a bug into the program as surely as a mistake in something 'non-trivial'."*

Survey participants from the two package management platforms have different perception about using trivial packages. Whereas most of the surveyed developers from npm (57.9%) do NOT believe that using trivial packages is a bad practice, the majority of the surveyed developers from PyPI (70.3%) consider using trivial packages as a bad practice.

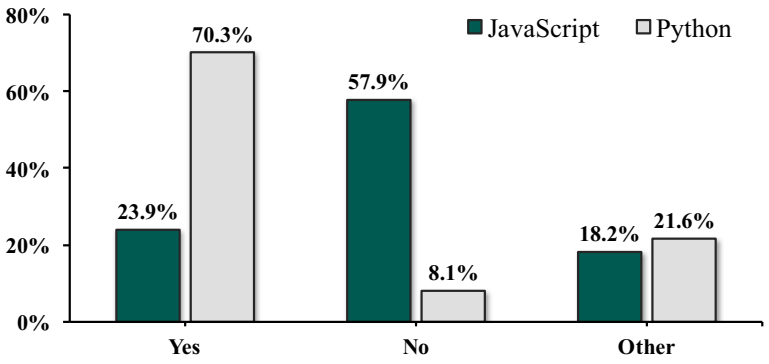


Fig. 4 Developer responses to the question "is using a trivial package bad?" Most JavaScript developers answered no, whereas most Python developers answered yes

5.2 Why Do Developers Use Trivial Packages?

While we have answered the question as to whether developers say 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. We then analyze separately the responses from the two surveys (JavaScript and Python). After gathering all of the responses, we grouped and categorized the responses in a two-phase iterative process. In the first phase, two of the authors carefully read the participant's answers and independently 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, the 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 independently 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 (Cohen 1960). 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 1, where a negative value means less than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement (Fleiss and Cohen 1973). In our categorization, the level of agreement measured between the authors was of 0.90 and 0.83 for the *npm* survey and *PyPI* survey, respectively, which is considered to be excellent inter-rater agreement.

Table 3 shows the reasons for using trivial packages, as reported by respondents from both JavaScript and Python surveys. As we can see from the table, the two most cited reasons

Table 3 Reasons for using trivial packages in *npm* and *PyPI*

Reason	Description	npm		PyPI	
		#Resp.	%	#Resp.	%
Well-implemented & tested	Participants state that trivial packages are effectively implemented and tested.	48	54.6%	20	54.1%
Increased productivity	Trivial packages reduce the time needed to implement existing source code.	42	47.7%	12	32.4%
Well-maintained code	It eases source code maintenance, since other developers maintain the trivial package.	8	9.1%	2	5.4%
Improved readability & reduced complexity	Using trivial packages improve the source code quality in terms of readability and reduce complexity.	8	9.1%	5	13.5%
Better performance	Trivial packages improve the performance of web applications compared to the use of large frameworks.	3	3.4%	0	0.0%
No reason	–	7	8.0%	7	18.9%

(i.e., well-implemented & tests and increased productivity) are the same for both *npm* and *PyPI* package management platforms. However, when it comes to the 3 less common reasons, there is a slight difference between *npm* and *PyPI*, most notably, the reason of trivial packages provide better performance was not evident in our survey.

Next, we discuss each of the reasons presented in Table 3 in more detail:

R1. **Well-implemented & tested:** 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 with 54.6% and 54.1% of the responses from JavaScript and Python, respectively. 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 P-*npm* 68, P-*npm* 4, and P-*PyPI* 5, who cite their reasons for using trivial packages as follows: P-*npm* 68: *"Tests already written, a lot of edge cases captured [...]."*, P-*npm* 4: *"There may be a more elegant/efficient/correct/cross-environment-complatible solution to a trivial problem than yours"*, and P-*PyPI* 5: *"They have covered extra cases that I would not do or thought initially."*

R2. **Increased productivity:** The second most cited reason is the improved productivity that using trivial packages enables with 47.7% and 32.4% for JavaScript and Python, respectively. 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 P-*npm* 13 and P-*npm* 27 from the JavaScript survey state: P-*npm* 13: *"[...] and it does save time to not have to think about how best to implement even the simple things."* & P-*npm* 27: *"Don't reinvent the wheel! if the task has been done before."*. Another example from the Python survey, participant P-*PyPI* 17 states: *"Often I do write the code myself. And then package it into a re-usable module so that I don't have to write it again later. And again. And again... At this point, whether the module is authored by myself or someone else is mostly irrelevant. What's relevant is that I get to avoid repeatedly implementing the same functionality for each new project."*

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:** A less common (9.1% and 5.4% of the responses from JavaScript and Python), 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, participants P-*npm* 45 and P-*PyPI* 1 states, P-*npm* 45: *"Also, a highly used trivial package is probable to be well maintained."* and P-*PyPI* 1: *"The simple advantages are that they may be trivial AND used by many people and therefore potentially maintained by developers."* 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 P-*npm* 80: *"[...], leveraging feedback from a larger community to fix bugs, etc."*

R4. **Improved readability & reduced complexity:** Participants also reported that using trivial packages improves the readability and reduces the complexity of their code

with 9.1% and 13% responses for the two package management platforms. For example, P-*npm* 34 states: *"immediate clarity of use and readability for other developers for commonly used packages[...]"* & P-*npm* 47 states: *"Simple abstract brings less complexity."* Python developers report the same advantage of using trivial packages. For example, P-*PyPI* 5 states that *"Code clarity. When many two liners become one liners it saves space. Its the whole point of batteries included mentally..."*

R5. **Better performance:** A few of the JavaScript participants (3.4%) stated that using trivial packages improves performance since it alleviates the need for their application to depend on large frameworks. Notably, the load time of trivial packages compared to larger JavaScript packages is small, which speeds up the overall load time of the applications. For example, P-*npm* 35 states: *"[...] you do not depend on some huge utility library of which you do not need the most part."* While JavaScript developers reported that trivial packages improve the performance, the Python developers do not report such a claim. One explanation for this is that JavaScript is used to develop front-end applications, which is often sensitive to performance i.e., load time, whereas the Python is used to implement applications in a wide variety of domains.

Overall the developer responses show that there is a different perception of using trivial package among developers from the two package management platforms. Only a small percentage (8.0%) of the respondents from JavaScript stated that they do not see a reason to use trivial packages. However, for Python developers 18.9% of the respondents believe that there are no advantages of using trivial packages.

JavaScript and Python developers believe that the two most cited reasons for using trivial packages are 1) they provide well implemented and tested code and 2) they increase productivity. However, only JavaScript developers cited that using trivial packages improves the performance of their applications. Additionally, only 8.0% of the JavaScript participants see no reason of using trivial packages while more than 18% of Python developers believe that there is no real reason to use trivial packages.

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 and 0.91 for *npm* and *PyPI*, respectively, which is considered to be an excellent agreement.

Table 4 lists the drawback mentioned by the survey respondents along with a brief description and the frequency of each drawback. As we can see from the table, the top two most cited drawbacks (i.e., dependency overhead and breakage of applications) are the same for both, *npm* and *PyPI*. However, for the less cited drawbacks, *npm* developers cited performance, development slow down and missed learning opportunities as the next set of drawbacks, whereas in *PyPI*, the developers consider security, development slow down and decreased performance as the next set of drawbacks. It is worth noting however that there is very little difference between the individual drawbacks (e.g., security vs. development

Table 4 Drawback of using trivial packages in *npm* and *PyPI*

Drawback	Description	npm		Python	
		#Resp.	%	#Resp.	%
Dependency overhead	Using trivial packages results in a dependency mess that is hard to update and maintain.	49	55.7%	25	67.6%
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%	12	32.4%
Decreased performance	Trivial packages decrease the performance of applications, which includes the time to install and build the application.	14	15.9%	3	8.1%
Slows development	Finding a relevant and high quality trivial package is a challenging and time consuming task.	11	12.5%	4	10.8%
Missed learning opportunities	The practice of using trivial packages leads to developers not learning and experiencing writing code for trivial tasks.	8	9.1%	0	0%
Security	Using trivial packages can open a door for security vulnerability.	7	8.0%	5	13.5%
Licensing issues	Using trivial packages could cause licensing conflicts.	3	3.4%	2	5.4%
No drawbacks	–	7	8.0%	3	8.1%

slow down) within the two package management platforms (i.e., *npm* and *PyPI*). Next, we discuss each of the drawbacks in more detail:

D1. Dependency overhead: 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 (Bogart et al. 2016; Mirhosseini and Parnin 2017). 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, which account for 55.7% of the responses form JavaScript developers. For example, P-*npm* 41 states: *"[...] people who don't actively manage their dependency versions could [be] exposed to serious problems [...]"* & P-*npm* 40: *"Hard to maintain a lot of tiny packages"*. For Python developers, the percentage of responses related to dependency overhead is high (67.6%) as well. Some example responses from Python developers that mention these issues are stated by participants P-*PyPI* 2, P-*PyPI* 4 & P-*PyPI* 13 who state that: P-*PyPI* 2: *"...it's more difficult to distribute something with a dependency that doesn't come with Python."*, P-*PyPI* 4: *"Lots of brittle dependencies."* & P-*PyPI* 13: *"When your projects consist of a lot trivial modules, it becomes almost impossible to track their update and some time you might forget what even they do."* Hence, while trivial packages may provide well-implemented/tested code and improve productivity, developers are clearly aware that the management of the additional dependencies is something they need to deal with.

D2. **Breakage of applications:** Developers also worry about the potential breakage of their application due to a specific package or version becoming unavailable. JavaScript developers stated this issue in 18.2% of the responses while the percentage is 32.4% for Python developers. For example, in the left-pad issue, the main reason for the breakage was the removal of left-pad, P-*npm* 4 states: *"Obviously the whole 'left-pad crash' exposed an issue"* & P-*PyPI* 22 states: *"potential for breaking (NPM leftpad situation)"*. However, since that incident, *npm* has disabled the possibility of a package being removed (npm Blog 2016). Although disallowing the removal solves part of the problem, packages can still be updated, which may break an application. This issue was clear from one of the responses, P-*PyPI* 7, who stated *"Potential for breaking changes from version to version."* 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.

D3. **Decreased performance:** This issue is related to the dependency overhead drawback. Developers mentioned that incurring the additional dependencies slowed down the build and run time and increased application installation times (15.9% and 8.1%). For example, P-*npm* 64 states: *"Too many metadata to download and store than a real code."* & P-*npm* 34 states: *"[...], slow installs; can make project noisy and unintuitive by attempting to cobble together too many disparate pieces instead of more targeted code."* Another Python developer ,P-*PyPI* 1, states: *"If the modules are not so ubiquitous, then needing the dependency is a real drag as one will have to install it. Also, the same job done with your own may run much faster and be easier to understand.* 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.

D4. **Slows development:** In some cases, the use of trivial packages may actually have a reverse effect and slow down development with 12.5% & 10.8% of responses from JavaScript and Python developers. For example, as P-*npm* 23 and P-*npm* 15 state: P-*npm* 23: *"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."* & P-*npm* 15: *"[...], 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. Python developers also agree on this issue, for example P-*PyPI* 15 states *"If finding, reading, and understanding the documentation of a module takes longer than reading its implementation, the hiding of functionality in third-part trivial modules obscures the source base."*

D5. **Missed learning opportunities:** In certain cases reported by only JavaScript developers (9.1%), the use of these trivial packages is seen as a missed learning opportunity for developers. For example, P-*npm* 24 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. In contrast to JavaScript developers, Python developers seem to not to be worried about this issue since the use of trivial packages is not as common within the Python developer community as JavaScript developers.

D6. **Security:** 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 (8.0% and 13.5%), for example, as P-*npm* 15 mentioned earlier, it is difficult to find

packages that are trustworthy. Also, P-*npm* 57 mentions: *"If you depend on public trivial packages then you should be very careful when selecting packages for security reasons"* & P-*PyPI* 3 states *"more dependencies, greater likelihood of not knowing of how code actually works at lower level, security issues."* 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.

D7. Licensing issues (3.4%): In some cases from both responses (3.4% and 5.4% for JavaScript and Python), developers are concerned about potential licensing conflicts that trivial packages may cause. For example, P-*npm* 73 states: *"[...], possibly license-issues"*, P-*npm* 62: *"[...], there is a risk that the 'trivial' package might be licensed under the GPL must be replaced anyway prior to shipping."* P-*PyPI* 23 also mentions *"Can be licensing hell."*

In general, we observe similar concerns regarding the use of trivial packages in the two software managements platforms studied. There were also approximately 8% of the responses in both package management platforms 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 Perceptions Under the Microscope

The developer surveys provided us with valuable insights on why developers use trivial packages and what they perceive to be their drawbacks. 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 developers' belief that trivial packages are well tested, and drawback, i.e., the impact of additional dependencies, based on our findings in Section 5.

6.1 Examining the 'Well Tested' Perception

As shown in Table 3, more than half of the responses from the studied package management platforms indicate that they use trivial packages because developers believe that they are well implemented and tested. However, is this really the case - are trivial packages really well tested? In this section, we want to examine whether this belief has any grounds or not.

6.1.1 Node Package Manager (*npm*)

npm requires that developers provide a test script name with the submission of their packages (listed in the package.json file). In fact, 73.7% (59,110 out of 80,232) 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 *npm* 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', which means that the trivial

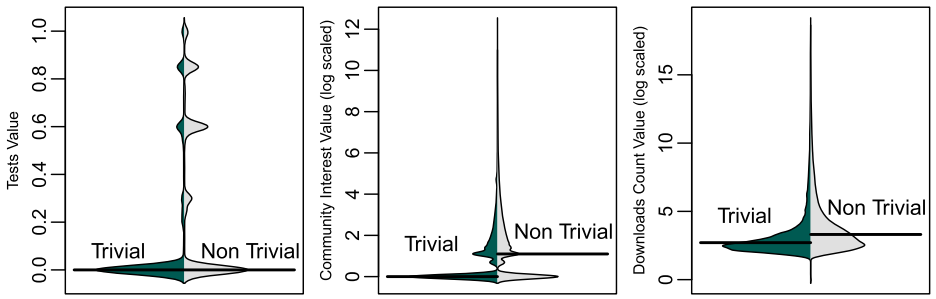
packages are used by many developers, we also consider the usage of a package as an indicator of it being well tested and implemented (Zambonini 2011). To carefully examine whether a package is really well tested and implemented, we use the *npm* online search tool (known as *npms* (Cruz and Duarte 2017)) 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 found 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 motivates us to investigate how well tested a trivial package is the response by P-*npm* 68, 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 P-*npm* 56, 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 P-*npm* 61 mentions: *"this code is tested and used by many, which makes it more trustful and reliable"*.

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 tests. We find that only 28.4% 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 5 shows the bean-plots for the Tests, Community interest and Download count. In all cases trivial packages have, on median, a smaller Community interest value and Download count compared to non-trivial packages except for the Tests value. The Fig. 5a shows that for the Tests metric, trivial packages have, on median, a similar value as non-trivial packages. That said, we observe from Fig. 5a 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. 0.30,

[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 (Cruz and Duarte 2017). To make our paper self-sufficient, we include how the metrics are calculated here.



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

Fig. 5 Distribution of Tests, Community Interest and Download Count Metrics for *npm* package management platform

0.6, 0.9 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 Grissom and Kim (2005), 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 5 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 Community interest and Download count values, their effect size is smaller than non-trivial packages.

> ***Contrary to developers' perception, only 28.4% of trivial JavaScript packages actually have tests.***

6.1.2 Python Package Index (*PyPI*)

Since *PyPI* does not collect any metadata to show if the Python package is tested or not, we use other data sources to examine the well tested perception. To do so, we use two ways to examine whether Python packages are tested or not: 1) we use the source code of the packages that are hosted on GitHub. 2) we relied on information about Python packages

Table 5 Mann-Whitney Test (p-value) and Cliff's Delta (d) for trivial vs. non trivial packages in *npm*

Metrics	p-value	d
Tests	2.2e-16	−0.222 (small)
Community interest	2.2e-16	−0.225 (small)
Downloads count	2.2e-16	−0.261 (small)

collected by the open source service libraries.io (https://libraries.io/). libraries.io monitors and collects the metadata of open source packages across 36 different package management platforms. It falls under the CC-BY-SA 4.0 licenses and has been used in other research work (e.g,. Decan et al. 2018a, b). We obtain the extracted metadata information related to *PyPI* package management. Once again, we examine the testing perception in three complementary ways.

1) **Tests:** we examine if the package has any test code written. Since there is no standard way to determine that a Python application has tests (e.g,. there exist more than 100 Python testing tools (https://wiki.python.org/moin/PythonTestingToolsTaxonomy)), we manually investigate whether the *PyPI* package contains test code written or not. The idea is that if the developers writes tests, then they will put these tests in the package repository. One example that motivated us to look for the test code of a package is the developer response: P-*PyPI* 11 who stated *"Shorter code overall, well-tested code for fundamental tasks helps smooth over language nits"*.

   Since this is a heavily manual process, we decide to examine a representative sample of the packages. Therefore, we take a statistically significant sample from the 6,759 Python packages that we identify as trivial Python packages (Section 4.1). The sample size is selected randomly to attain 5% confidence interval and a 95% confidence level. This sampling process result in 364 *PyPI* trivial packages. Then, two of the authors manually examine the code bases of sampled packages looking for test code to identify the packages that has test. After that, we measure Cohen's Kappa coefficient to evaluate the level of agreement between the two annotators (Cohen 1960). As a result of this process, we find that the level of agreement between the two authors to be 0.97, which is consider to be excellent agreement. Finally, the two authors discuss the cases that they do not agree on and come to an agreement.
2) **Community interest:** evaluates the community interest in the packages, using the number of stars on GitHub, forks, subscribers and contributors. We adopted the same formula defined by *npms*, which is basically 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.
3) **Usage count:** represents the number of applications that use a package. The more applications using a Python package, the more popular that package is. This may also indicate that the package is of high quality. For example, P-*PyPI* 11 indicated *"The simple advantages are that they may be trivial AND used by many people and therefore potentially maintained by developers."* Hence, we use the *usage count* metric since it indicates the package quality; thus, many developers use it in their applications. To calculate the number of Python applications that use *PyPI* trivial packages, we use the libraries.io dataset that provides a list of Python applications and the packages they depend on. Also, for each *PyPI* package in our dataset, we count the number of Python applications that use that package.

We found that out of the 364 sampled trivial Python packages that we manually examined, 185 (50.82%) packages do not have test code in them, while 179 (49.18%) of the examined packages have test code written in them. It is important to note that our analysis only examines whether a trivial package has tests or not, whether these tests are actually effective is a completely different issue and is one of the reasons for examining the other two metrics Community interest and Usage count.

Figure 6 shows the bean-plots for the Community interest and Usage count values for trivial and non-trivial Python packages in our dataset. The figures show that in the two cases trivial Python packages have, on median, a smaller Community interest value and Usage count compared to non-trivial packages. That said, we observe from Fig. 6a that in the case of the Community interest metric, we see clearly the median values are lower for trivial packages. Figure 6b shows that the distribution of the Usage count metric is similar for both, trivial and non-trivial packages. Once again, we 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. We also use Cliff's Delta (d) to measure the effect size between *PyPI* trivial and non-trivial packages. Table 6 shows the p-values and effect size values. We observe that in the cases of community interest and usage count, the differences are statistically significant, and the effect size is small and negligible, respectively.

> ***Only 49.2% of trivial PyPI packages actually have tests. Also, in terms of Community interest and Usage count, there is only a small to negligible difference between PyPI trivial and non-trivial packages.***

6.2 Examining the 'Dependency Overhead' Perception

As discussed in Section 5, the top cited drawback of using trivial packages is 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,. de Souza and Redmiles 2008; Decan et al. 2016; Abate et al. 2009) that can be examined in a multitude of ways. We choose to examine the issue from both, the application and the package perspectives.

6.2.1 Application-level Analysis

When compared to coding trivial tasks themselves, using a trivial package imposes extra dependencies. One of the most problematic aspects of managing dependencies for

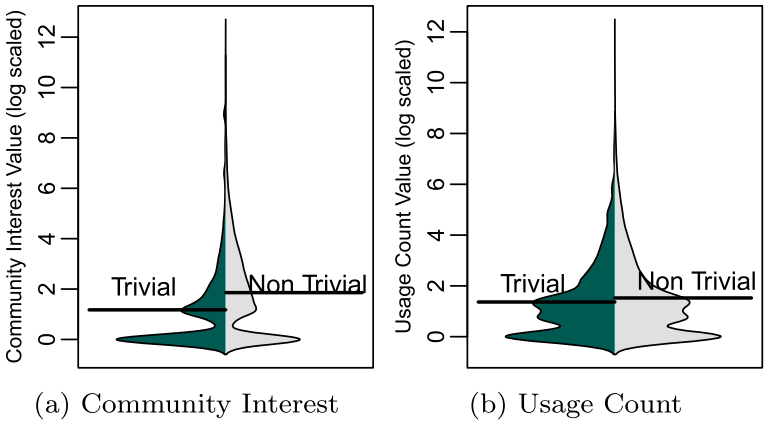


Fig. 6 Distribution of Community Interest and Usage Count Metrics for *PyPI* package management platform

Table 6 Mann-Whitney Test (p-value) and Cliff's Delta (d) for trivial vs. non trivial packages in *PyPI*

Metrics	p-value	d
Community interest	2.2e-16	−0.251 (small)
Usage count	0.004557	−0.039 (negligible)

applications is when these dependencies are updated, 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 ensure the proper integration of new releases. The bean-plots in Figs. 7 & 8 show the distribution of the number of releases for our studied package management platforms. Figure 7a shows that trivial packages on *npm* have less releases than non-trivial packages (median is 1 for trivial and 2 for non-trivial packages). However, when we examine the number of different release types, we found that trivial and non-trivial *npm* packages have similar numbers of minor and major releases (Fig. 7c & b). As for the patch releases, trivial *npm* packages have less patch releases. In Fig. 8a, we also observe that trivial packages on *PyPI* have less releases than non-trivial packages. We again examine the number of releases of *PyPI* packages based on the release type. Figures 8b, c, and d show the distribution of minor, major, and patch releases for trivial and non-trivial *PyPI* packages. From Fig. 8b and c, we do not see any difference between trivial and non-trivial packages for the minor and major releases. As for the patch releases, we observe that trivial *PyPI* packages have a smaller number of patch releases. 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. In addition, to examine whether the differences in the distribution of the type of releases between trivial and non-trivial packages are statistically significant, we performed a Wilcox test. We also use Cliff's Delta (d) to examine the effect size. Table 7 shows the p-values and the effect size for all the releases types for *npm* and *PyPI*. It shows that for all the releases types the differences are statistically significant, having p-values < 0.05. Also, the effect size values are small or negligible.

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. For example in the JavaScript application that use *npm* packages, 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., P-*npm* 8: *"[...] Also, people who don't lock*

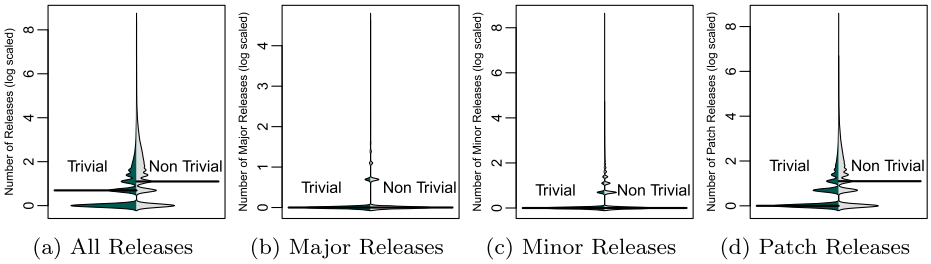
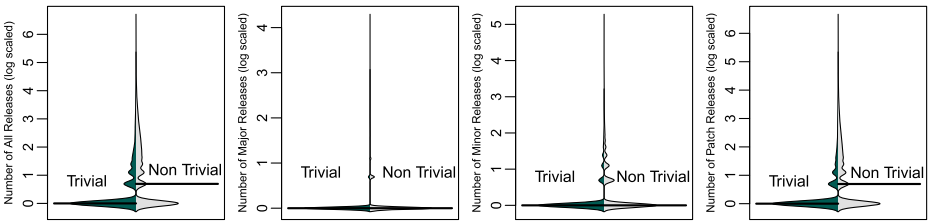


Fig. 7 Distribution of different types of releases for trivial and non-trivial *npm* packages



(a) All Releases (b) Major Releases (c) Minor Releases (d) Patch Releases

Fig. 8 Distribution of different types of releases for trivial and non-trivial *PyPI* packages

down their versions are in for some pain". In general, 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 a configuration file next to every package name for example *npm* defines it in the packages.json file. We examined the frequency at which trivial and non-trivial packages are locked. For *npm*, we find that on average, trivial packages are locked 26.3% of the time, whereas non-trivial packages are locked 28.2% of the time. The Wilcox test also shows that the difference is statistically significant p-value < 0.05 (p-value $=$ 9.116e-07). On the other hand, in *PyPI*, we find that on average, trivial packages are locked 31.7% of the time, whereas non-trivial packages are locked 36.2% of the time. Also, the Wilcox test shows that the difference is statistically significant with p-value $=$ 9.707e-08.

Our findings show that trivial packages are locked less in *npm* and the same is true in *PyPI* where trivial packages are locked less than non-trivial packages. In both cases however, we find that there is not a large difference between the percentage of packages (trivial vs. non-trivial) being locked.

6.2.2 Package-level Analysis

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 on *npm*, 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`

Table 7 Mann-Whitney Test (p-value) and Cliff's Delta (d) of the release type for trivial vs. non trivial packages for *npm* and *PyPI*

Release type	npm		PyPI	
	p-value	d	p-value	d
All	2.2e-16	−0.2016 (small)	2.2e-16	−0.2995 (small)
Minor	2.2e-16	−0.0823 (negligible)	2.2e-16	−0.2447 (small)
Major	2.2e-16	−0.1185 (negligible)	2.2e-16	−0.1276 (negligible)
Patch	2.2e-16	−0.1985 (negligible)	2.2e-16	−0.2729 (small)

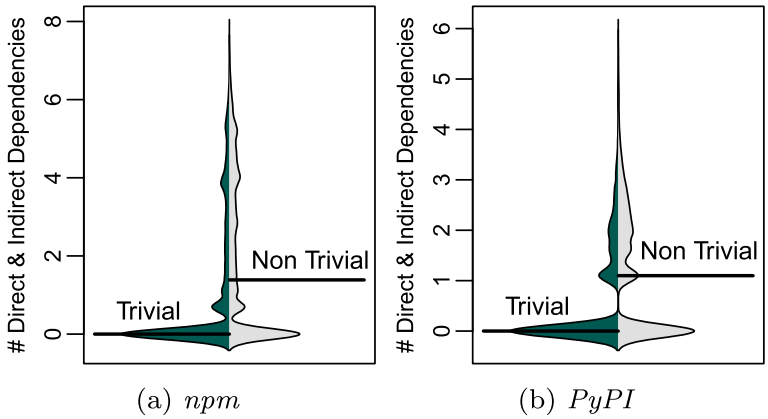


Fig. 9 Distribution of direct & indirect dependencies for trivial and non-trivial packages (log scale). For *npm* (p-value < 2.2e-16 & Cliff's Delta (d) -0.238 (small)) while *PyPI* (p-value < 2.2e-16) & Cliff's Delta (d) -0.246 (small)

file and the `require` statements will provide the direct dependencies, but not the indirect dependencies. Hence, we downloaded all the packages in our *npm* dataset, mock installed[4] them and build the dependency graph for the *npm* platform.

Similarly, for *PyPI*, we count the actual number of (direct and indirect) dependencies that the package requires. To do so, we leveraged the metadata provided by Valiev et al. (2018). In their study, Valiev et al. extracted the list of direct and indirect dependencies of each package on *PyPI*. We resort to use the data provided in Valiev et al. (2018) since it is recently extracted data and covers the history of *PyPI* for more than six years. We then read the dependencies of each package and build a dependency graph for the *PyPI* platform.

Figure 9 shows the distribution of dependencies for trivial and non-trivial packages for the *npm* and *PyPI*. Since most trivial packages have no dependencies, the median is zero. Therefore, we bin the trivial packages based on the number of their dependencies and calculate the percentage of packages in each bin.

Table 8 shows the percentage of packages and their respective number of dependencies for both *npm* and *PyPI*. We observe that the majority of *npm* trivial packages (56.9%) have zero dependencies, 21% have between 1-10 dependencies, 3.8% have between 11-20 dependencies, and 18.4% have more than 20 dependencies. The table also shows that *PyPI* trivial packages do not have as much dependencies as the *npm* packages. In fact 63.2% of *PyPI* packages have zero dependencies and approx. 34% of trivial packages have between 1-20 dependencies. Only approx. 3% of the *PyPI* trivial packages have more than 20 dependencies. Interestingly, the table shows that some of the trivial packages in *npm* have many dependencies, which indicates that indeed, trivial packages can introduce significant dependency overhead. It also shows that *PyPI* trivial packages have small number of dependencies. One explanation of such a difference is that Python language has a more mature standard API that provides most of the needed utility functionalities.

[4] we modified the *npm* code to intercept the install call and counted the installations needed for every package.

Table 8 Percentage of packages vs. the number of dependencies used in the *npm* and *PyPI* package management platforms

Packages	npm # Dependencies (Direct & Indirect)				PyPI # Dependencies (Direct & Indirect)			
	0	1-10	11-20	>20	0	1-10	11-20	>20
Trivial	56.9%	21%	3.8%	18.4%	63.2%	29.6%	4.3%	2.9%
Non Trivial	37.1%	24.1%	6.8%	32.1%	42.5%	39.4%	10.7%	7.4%

Trivial packages have fewer releases and 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 43.4% of npm trivial packages have at least one dependency and 18.4% of npm trivial packages have more than 20 dependencies while 36.8% of PyPI trivial packages have at least one dependency and 2.9% of PyPI 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 (P-*npm* 39) 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 studies, 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 125 JavaScript and Python respondents, 81 (aprox. 65%) 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 JavaScript and Python development communities.

7.2 Implications of Our Study

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

7.2.1 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 a bad practice, especially JavaScript developers. 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. For example P-*npm* 15 indicated that *"... we have the problem of locating packages that are both useful and 'trustworthy' in an ever growing sea of packages."* 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. For example, P-*npm* 35 *"especially in JavaScript relieves you from thinking about cross browser compatibility for special cases/coming up with polyfills and testing all edge cases yourself. Basically it's a substitute for the missing standard library. And you do not depend on some huge utility library of which you do not need the most part"* & P-*PyPI* 23 *"Usually an indication of the inadequacy of the standard library. This seems particularly so of JavaScript where you might find yourself using many such modules."* Therefore, there is a need by the JavaScript community to create a standard JavaScript API or library in order to reduce the dependence on trivial packages. This issue of creating such a standard JavaScript library is under much debate (Fuchs 2016).

7.2.2 Implications for Future Research

Our study mostly focused on determining the prevalence, reasons for and drawbacks of using trivial packages in two large package management platforms *npm* and *PyPI*. Based on our findings, we find a number of implications and 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 application. 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 application. As some of the survey respondents pointed out, they are much more likely to use trivial packages early on in the development life cycle, so they do not waste time on trivial tasks and focus on the more fundamental tasks of their application. Once their application matures, they start to look for ways to reduce dependencies since they pose potential points of failure for their application. Our study motivates future work to examine the relationship between team experience and application 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 Questions & Answers (Q&A) sites such as StackOverflow or Reddit. For example, P-*npm* 84 stated that *"I'd have to do research on how to solve a particular problem, peruse questions and answers on StackOverflow, Reddit, or Coderanch, and find the most recent and readable solution among everything I've found, then write it myself. Why go through all of this work when you can simply 'require()' someone else's solution and continue working towards your goal in a matter of seconds?"* 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* and/or *PyPI* is seen as 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 (Abdalkareem et al. 2017a, b; Wu et al. 2018; Baltes and Diehl 2018), we are not aware of any studies that compare code reuse from Q&A sites and trivial packages. Our findings indicate the need for such a study.

8 Related Work

In this section, we discus the work that is related to our study. We divided the related work to work related to code reuse in general and work studied software ecosystems.

8.1 Studies of Code Reuse

Prior research on code reuse has shown its many benefits, which include improving quality, development speed, and reducing development and maintenance costs (Mockus 2007; Lim 1994; Mohagheghi et al. 2004; Basili et al. 1996). For example, Sojer and Henkel (2010) 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. (2008) 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 (2007) 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 (Di Cosmo et al. 2011). Furthermore, a bug in the reused component could propagate to the target system (Dogguy et al. 2011). 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 JavaScript and Python applications.

8.2 Studies of Software Ecosystems

In recent years, analyzing the characteristics of ecosystems in software engineering has gained momentum (Bavota et al. 2013; Bloemen et al. 2014; Manikas 2016; Decan et al. 2016). For example, in a recent study, Bogart et al. (2015) and Bogart et al. (2016) empirically studied three ecosystems, including *npm*, and found that developers struggle with changing versions as they might break dependent code. Wittern et al. (2016) 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 are steadily growing. More than 80% of packages have at least one direct dependency.

Other studies examined the size characteristics of packages in an ecosystem. German et al. (2013) 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 (2011) analyzed the Ruby ecosystem and found that many small and large projects are interconnected. Decan et al. (2018b) investigated the evolution of package dependency networks for seven packaging ecosystems. Their findings reveal that the studied packaging ecosystems grow over time in term of number of published and updated packages. They also observed that there is an increasing number of transitive dependencies for some packages.

Other works investigate the challenges of using external packages of a software ecosystem including; identify conflicts between JavaScript package (Patra et al. 2018), examine how pull requests help developers to upgrade out-of-date dependencies in their applications (Mirhosseini and Parnin 2017), study the usage of repository badges in the *npm* ecosystem (Trockman et al. 2018), and the usage of dependency graph to discover hidden trend in an ecosystem (Kula et al. 2018).

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 and we studied them in two different package management platforms *npm* and *PyPI*. 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 (Hunter 2001; Seaman 1999). 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 65% of the participants said they wanted to know our study outcomes.

9 Threats to Validity

In this section, we discuss the threats to the validity of our case study.

9.1 Internal Validity

Internal validity concerns factors that may have influenced our results such as our datasets collection process. To study the reasons for and drawback of using trivial packages, we surveyed developers. 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 and we publicly share our survey and all of our anonymized survey responses (Abdalkareem et al. 2019). Moreover, the way we asked the survey questions might have affected the response from our respondents, causing their responses to advocate or not advocate the use of trivial packages. To reduce this bias, we ensure participants' anonymity. Also, our study may be impacted by the fact that an overlap does not exist between the developer groups who participated in the two user studies (i.e., defining trivial packages and understanding developers' perception about the use of trivial packages). We find that the second survey served as a confirmation of the observations made by the first survey participants, however, given that these are two different populations, they may have reported on different observations.

We removed test code from our dataset to ensure that our analysis only considers production source code. We identified test code by searching for the term 'test' (and its variants e.g., 'TEST_code') in the file names and file paths. Even though this technique is widely accepted in the literature (Gousios et al. 2014; Tsay et al. 2014; Zhu et al. 2014), 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. We found that in all the examined cases contain test code.

In addition, to examine the well-tested perception for the *PyPI* trivial packages, the first two authors manually examined the source code of the trivial packages to classify whether they have test code written or not. To ensure the validity of our classification, we measure the classification agreement between the two authors. We found that the classification agreement between the two authors to be excellent (Cohen's Kappa value of 0.97).

9.2 Construct 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 and 13 Python developers. However, we find that there was consensus for what is considered a trivial package. Although our analysis shows that packages with $\leq$ 35 LOC and a complexity $\leq$ 10 are trivial packages, we believe that other definitions are possible for trivial packages. That said, of the 125 survey participants that we emailed about using trivial packages, only 2 mentioned that a 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 cases, although clearly it is not perfect.

In addition, to determine what is considered to be a trivial package, we conducted an experiment with JavaScript and Python developers who are mostly students (undergraduate and graduate students) with some professional experience. While this may not present professional developers per se (Sjoberg et al. 2002), prior work has shown that experiment with students will provide the same results as professional developers in software engineering domain (Salman et al. 2015; Höst et al. 2000).

To identify the JavaScript and Python applications that we examine in our study, we rely on the metadata provided by the GHTorrent dataset (Gousios et al. 2014). Thus, our selection of JavaScript and Python applications heavily depends on the correctness of the applications' programming language listed in GHTorrent.

We use the LOC and cyclomatic 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. Our experience tells us that most developers only look at the package itself and not at its dependencies when determining if it is trivial or not. That said, when we replicated this questionnaire with another set of participants from the Python language community, we found that developers seem to confirm our definition of trivial JavaScript/Python packages (Abdalkareem et al. 2019).

Based on our user study, we defined trivial npm packages as a package that have <= 35 LOC and Cyclomatic Complexity <= 10. However, one threat to this definition is that 10 cyclomatic complexity is high for a package to be trivial. To examine this concern, we calculate the cyclomatic complexity of all the non-trivial packages in our dataset and found that on average non-trivial npm packages have a cyclomatic complexity of 803, which indicates

that 10 Cyclomatic complexity value in our definition is still significantly smaller compared to the one for non-trivial packages.

To study trivial packages in the *PyPI* package management platform, we were able to extract 63,912 packages. Collecting more packages may provide more details about trivial packages on the *PyPI* package management platform. Also, to identify the Python applications that use *PyPI* trivial packages, we use the `snakefood` tool (http://furius.ca/snakefood/) to extract the applications dependencies. Hence, we are limited by the accuracy of `snakefood` in extracting the used packages in Python applications.

In our study, to understand why developers use trivial packages, we conducted two user surveys with JavaScript and Python developers. These two surveys were performed on different dates, and as a consequence, may affect the outcome of the survey results. However, given that these two package management platforms are independent, we envision that the impact of this date shift is not significant.

In our study, to identify developers who used trivial packages in their applications, we use regular expressions to identify these packages. This process may flag the wrong package by the developers. To mitigate this threat, during our analysis, we make sure that we extract the right packages through several rounds of manual checking of the results. In addition, none of the developers that we contacted indicated that she/he does not use the identified packages, which serves as a slight confirmation that our methodology is not incorrect.

In our study on *npm*, 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 *npms* metrics. We also use libraries.io to calculate the community interested and the usage count metrics for *PyPI* packages, and our measurements are as accurate as libraries.io. We resort to use the libraries.io data since it has been used on other prior work (e.g,. Decan et al. 2018a, b). In addition, we use the dataset provided by Valiev et al. (2018) to measure the direct and indirect dependencies of the packages on *PyPI*.

In our analysis, we also use different R packages to perform our analysis, our analysis may be impacted by the accuracy of these used R packages. To mitigate this threat we make our dataset and used tools available online (Abdalkareem et al. 2019).

9.3 External Validity

External validity considers the generalization of our findings. All of our findings were derived from open source JavaScript applications and *npm* packages and its replication on Python and *PyPI* packages. Even though we believe that the two studied package management platforms are amongst the most commonly used ones, 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 (Flyvbjerg 2006). We believe that strong empirical evidence is built from both studies on individual cases and studies on large samples.

Our list of reasons for and drawbacks of using trivial packages are based on a survey of 88 JavaScript and 37 Python developers. Although this is a large number of developers, our results may not hold for all 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 of them are experienced developers.

We do not distinguish between the domain of studied packages, which may impact the findings. However, to help mitigate any bias we analyzed more than 500,000 *npm* and

74,663 *PyPI* packages that cover a wide range of package domains. Lastly, our study is based on open source applications that are hosted on GitHub, therefore, our study may not generalize to other open source or commercial applications.

10 Conclusion

The use of trivial packages is an increasingly popular trend in software development (Abdalkareem et al. 2017; Abdalkareem 2017). Like any development practice, it has its proponents and opponents. The goal of our study is to extend our understanding of the use of trivial packages. We examine the prevalence, reasons, and drawbacks of using trivial packages in different package management platforms. Thus, we consider trivial packages in *PyPI* in addition to the previous studied *npm* (Abdalkareem et al. 2017).

Our results indicate that trivial packages are commonly and widely used in JavaScript and Python applications. We also find that while the majority of JavaScript developers in our study do not oppose the use of trivial packages, the majority of Python developers believe that using trivial packages could be harmful. Additionally, based on the developers' responses, developers from the two package management platforms stated that the main reasons for developers to use trivial packages is due to the fact that they are considered to be well implemented and tested. They do cite the additional dependencies' overhead as a drawback of using these trivial packages. Our empirical study showed that considering trivial packages to be well tested is a misconception since more than half of the studied trivial package do not even have tests. However, these trivial packages seem to be 'deployment tested' and have similar Community interest and Download/Usage count values as non-trivial packages. In addition, we find that some of the trivial packages have their own dependencies. In our studied dataset, 18.4% of the *npm* and 2.9% of the *PyPI* trivial packages have more than 20 dependencies. Hence, developers should be careful about which trivial packages they use.

Based on our findings, we provide the following practical suggestions for software developers:

- Developers should not assume that trivial packages are well-tested and implemented since we found only 28.4% and 49.2% of *npm* and *PyPI* trivial packages have test code.
- Due to the fact that trivial packages have their own dependencies, developers should be aware that using these trivial packages would increase the dependency overhead of their applications.

Acknowledgments The authors are grateful to the many survey respondents who dedicated their valuable time to respond to our surveys. Also, the authors would like to thank the anonymous reviewers and the editor for their thoughtful feedback and suggestions that help us improve our study.

References

Abate P, Di Cosmo R, Boender J, Zacchiroli S (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, pp 89–99

Abdalkareem R (2017) Reasons and drawbacks of using trivial npm packages: The developers' perspective. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, ACM, pp 1062–1064

Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why do developers use trivial packages? an empirical case study on npm. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE '17, ACM, pp 385–395

Abdalkareem R, Oda V, Mujahid S, Shihab E (2019) On the impact of using trivial packages: An empirical case study on npm and pypi. https://doi.org/10.5281/zenodo.3095009

Abdalkareem R, Shihab E, Rilling J (2017) On code reuse from Stack Overflow : An exploratory study on Android apps. Inf Softw Technol 88(C):148–158

Abdalkareem R, Shihab E, Rilling J (2017) What do developers use the crowd for? a study using Stack Overflow. IEEE Softw 34(2):53–60

Baltes S, Diehl S (2018) Usage and attribution of Stack Overflow code snippets in gitHub projects. Empirical Software Engineering

Basili VR, Briand LC, Melo WL (1996) How reuse influences productivity in object-oriented systems. Commun ACM 39(10):104–116

Bavota G, Canfora G, Penta MD, Oliveto R, Panichella S (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, pp 280–289

Blais M snakefood: Python Dependency Graphs. http://furius.ca/snakefood/. (accessed on 09/23/2018)

Bloemen R, Amrit C, Kuhlmann S, Ordóñez Matamoros G (2014) Gentoo package dependencies over time. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR '14, ACM, pp 404–407

Bogart C, Kastner C, Herbsleb J (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, pp 86–89

Bogart C, Kästner C, Herbsleb J, Thung F (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, pp 109–120

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

Castelluccio M, An L, Khomh F (2019) An empirical study of patch uplift in rapid release development pipelines. Empir Softw Eng 24(5):3008–3044

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas 20:37–46

Cruz A, Duarte A (2017) npms. https://npms.io/. (accessed on 02/20/2017)

de Souza CRB, Redmiles DF (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, pp 241–250

Decan A, Mens T, Constantinou E (2018a) On the impact of security vulnerabilities in the npm package dependency network. In: International Conference on Mining Software Repositories

Decan A, Mens T, Grosjean P (2018b) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering

Decan A, Mens T, Grosjean P et al (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, volume 1 of SANER '16, IEEE, pp 493–504

Di Cosmo R, Di Ruscio D, Pelliccione P, Pierantonio A, Zacchiroli S (2011) Supporting software evolution in component-based FOSS systems. Sci Comput Program 76(12):1144–1160

Dogguy M, Glondu S, Le Gall S, Zacchiroli S (2011) Enforcing type-Safe linking using inter-package relationships. Studia Informatica Universalis 9(1):129–157

Ebert C, Cain J (2016) Cyclomatic complexity. IEEE Softw 33(6):27–29

Fleiss JL, Cohen J (1973) The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. Educ Psychol Meas 33:613–619

Flyvbjerg B (2006) Five misunderstandings about case-study research. Qual Inq 12(2):219–245

Fuchs T (2016) What if we had a great standard library in JavaScript? – medium. https://medium.com/@thomasfuchs/what-if-we-had-a-great-standard-library-in-javascript-52692342ee3f.pw7d4cq8j. (accessed on 02/24/2017)

German D, Adams B, Hassan A (2013) Programming language ecosystems: the evolution of R. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering, CSMR '13, IEEE, pp 243–252

Gousios G, Vasilescu B, Serebrenik A, Zaidman A (2014) Lean ghtorrent: Github data on demand. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR '14, ACM, pp 384–387

Grissom RJ, Kim JJ (2005) Effect sizes for research: A broad practical approach. Lawrence Erlbaum Associates Publishers

Haefliger S, Von Krogh G, Spaeth S (2008) Code reuse in open source software. Manag Sci 54(1):180–193
Haney D (2016) Npm & left-pad: Have we forgotten how to program? http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/. (accessed on 08/10/2016)
Harris R (2015) Small modules: it's not quite that simple. https://medium.com/@Rich_Harris/small-modules-it-s-not-quite-that-simple-3ca532d65de4. (accessed on 08/24/2016)
Hemanth HM (2015) One-line node modules -issue#10- sindresorhus/ama. https://github.com/sindresorhus/ama/issues/10. (accessed on 08/10/2016)
Höst M, Regnell B, Wohlin C (2000) Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. Empir Softw Eng 5(3):201–214
Hunter JE (2001) The desperate need for replications. J Consum Res 28(1):149–158
Inoue K, Sasaki Y, Xia P, Manabe Y (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, pp 331–341
Kabbedijk J, Jansen S (2011) Steering insight: An exploration of the Ruby software ecosystem. In: Proceedings of the Second International Conference of Software Business, ICSOB '11, Springer, pp 44–55
Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining gitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR '14, ACM, pp 92–101
Kula RG, Roover CD, German DM, Ishio T, Inoue K (2018) A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, volume 00 of SANER '18, pp 288–299
Libraries.io. Libraries.io - the open source discovery service. https://libraries.io/. (accessed on 05/20/2018)
Libraries.io (2017) Pypi. https://libraries.io/pypi. (accessed on 03/08/2017)
Lim WC (1994) Effects of reuse on quality, productivity, and economics. IEEE Softw 11(5):23–30
Macdonald F (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. (accessed on 08/24/2016)
Manikas K (2016) Revisiting software ecosystems research: a longitudinal literature study. J Syst Softw 117:84–103
McCamant S, Ernst MD (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, pp 287–296
Mirhosseini S, Parnin C (2017) Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE '17, IEEE Press, pp 84–94
Mockus A (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, p 7–
Mohagheghi P, Conradi R, Killi OM, Schwarz H (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, pp 282–292
npm (2016) What is npm? — node package managment documentation. https://docs.npmjs.com/getting-started/what-is-npm. (accessed on 08/14/2016)
npm Blog T (2016) The npm blog changes to npm's unpublish policy. http://blog.npmjs.org/post/141905368000/changes-to--unpublish-policy. (accessed on 08/11/2016)
Orsila H, Geldenhuys J, Ruokonen A, Hammouda I (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, pp 159–170
Patra J, Dixit PN, M. Pradel (2018) Conflictjs: Finding and understanding conflicts between javaScript libraries. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, pp 741–751
Python Python testing tools taxonomy - python wiki. https://wiki.python.org/moin/PythonTestingToolsTaxonomy. (accessed on 05/16/2018)
Rahman MT, Rigby PC, Shihab E (2019) The modular and feature toggle architectures of google chrome. Empir Softw Eng 24(2):826–853
Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in gitHub. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '14, ACM, pp 155–165

Salman I, Misirli AT, Juristo N (2015) Are students representatives of professionals in software engineering experiments? In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1 of ICSE '15, . IEEE, pp 666–676

SciTools Understand tool. https://scitools.com/. (accessed on 04/16/2019)

Seaman CB (1999) Qualitative methods in empirical studies of software engineering. IEEE Trans Softw Eng 25(4):557–572

Singer J, Sim SE, Lethbridge TC (2008) Software engineering data collection for field studies. In: Guide to Advanced Empirical Software Engineering. Springer, london, pp 9–34

Sjoberg DIK, Anda B, Arisholm E, Dyba T, Jorgensen M, Karahasanovic A, Koren EF, Vokac M (2002) Conducting realistic experiments in software engineering. In: Proceedings International Symposium on Empirical Software Engineering, IEEE, pp 17–26

Sojer M, Henkel J (2010) Code reuse in open source software development Quantitative evidence, drivers, and impediments. J Assoc Inf Syst 11(12):868–901

Trockman A, Zhou S, Kästner C, Vasilescu B (2018) Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In: Proceedings of the International Conference on Software Engineering, ICSE '18, ACM

Tsay J, Dabbish L, Herbsleb J (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, pp 356–366

Valiev M, Vasilescu B, Herbsleb J (2018) Ecosystem-level determinants of sustained activity in open-source projects A case study of the pyPi ecosystem. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '18. ACM

Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in gitHub. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE '15, ACM, pp 805–816

Williams C (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. (accessed on 08/24/2016)

Wittern E, Suter P, Rajagopalan S (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, pp 351–361

Wu Y, Wang S, Bezemer C-P, Inoue K (2018) How do developers utilize source code from Stack Overflow? Empirical Software Engineering

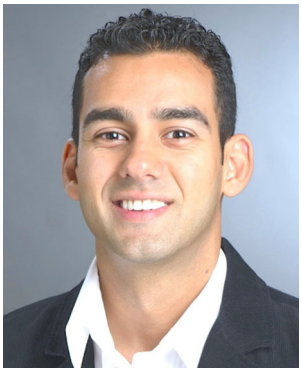
Zambonini D (2011) A Practical Guide to Web App Success, chapter 20. Five Simple Steps. (accessed on 02/23/2017). In: Gregory O (ed)

Zhu J, Zhou M, Mockus A (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, pp 30:1–30:4

Rabe Abdalkareem is a postdoctoral fellow in the Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada. He received his Ph.D. in Computer Science and Software Engineering from Concordia University, Montreal, Canada. His research investigates how the adoption of crowdsourced knowledge affects software development and maintenance. Abdalkareem received his masters in applied Computer Science from Concordia University. His work has been published at premier venues such as FSE, ICSME and MobileSoft, as well as in major journals such as TSE, IEEE Software, EMSE and IST. Contact him at rab_abdu@encs.concordia.ca; http://users.encs.concordia.ca/rababdu.

Vinicius Oda is a MASc. student in the Department of Computer Science and Software Engineering at Concordia University, Montreal. His research interests includes Software Engineering, Software Ecosystems, and Mining Software Repositories, among others.

Suhaib Mujahid is a Ph.D. student in the Department of Computer Science and Software Engineering at Concordia University. He received his masters in Software Engineering from Concordia University (Canada) in 2017. He obtained his Bachelors in Information Systems at Palestine Polytechnic University. His research interests include wearable applications, software quality assurance, mining software repositories and empirical software engineering. You can find more about him at http://users.encs.concordia.ca/smujahi.



Emad Shihab is an Associate Professor and Concordia University Research Chair in the Department of Computer Science and Software Engineering at Concordia University. His research interests are in Software Engineering, Mining Software Repositories, and Software Analytics. His work has been published in some of the most prestigious SE venues, including ICSE, ESEC/FSE, MSR, ICSME, EMSE, and TSE. He serves on the steering committees of PROMISE, SANER and MSR, three of the leading conferences in the software analytics areas. His work has been done in collaboration with and adopted by some of the biggest software companies, such as Microsoft, Avaya, BlackBerry, Ericsson and National Bank. He is a senior member of the IEEE. His homepage is: http://das.encs.concordia.ca.

Affiliations

Rabe Abdalkareem[1] · Vinicius Oda[1] · Suhaib Mujahid[1] · Emad Shihab[1]

Vinicius Oda
v_oda@encs.concordia.ca

Suhaib Mujahid
s_mujahi@encs.concordia.ca

Emad Shihab
eshihab@encs.concordia.ca

[1] Data-Driven Analysis of Software (DAS) Lab, Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada