Empirical Software Engineering (2021) 26: 27
https://doi.org/10.1007/s10664-020-09904-w

Helping or not helping? Why and how trivial packages impact the *npm* ecosystem

Xiaowei Chen[1] · Rabe Abdalkareem[2] · Suhaib Mujahid[1] · Emad Shihab[1] · Xin Xia[3]

Accepted: 13 November 2020 / Published online: 2 March 2021

Abstract

Developers often share their code snippets by packaging them and making them available to others through software packages. How much a package does and how big it is can be seen as positive or negative. Recent studies showed that many packages that exist in the *npm* ecosystem are trivial and may introduce high dependency overhead. Hence, one question that arises is why developers choose to publish these trivial packages. Therefore, in this paper, we perform a developer-centered study to empirically examine why developers choose to publish such trivial packages. Specifically, we ask 1) why developers publish trivial packages, 2) what they believe to be the possible negative impacts of these packages, and 3) how such negative issues can be mitigated. The survey response of 59 JavaScript developers who publish trivial *npm* packages showed that the main advantages for publishing these trivial packages are to provide *reusable components*, *testing & documentation*, and *separation of concerns*. Even the developers who publish these trivial packages admitted to having issues when they publish such packages, which include the maintenance of multiple packages, dependency hell, finding the right package, and the increase of duplicated packages in the ecosystems. Furthermore, we found that the majority of the developers sug-

Communicated by: Massimiliano Di Penta

✉ Xiaowei Chen
c_iaowei@encs.concordia.ca

Rabe Abdalkareem
abdrabe@gmail.com

Suhaib Mujahid
s_mujahi@encs.concordia.ca

Emad Shihab
eshihab@encs.concordia.ca

Xin Xia
xin.xia@monash.edu

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

[2] Software Analysis and Intelligence Lab (SAIL), Kingston, Canada

[3] Faculty of Information Technology, Monash University, Clayton, Australia

gested grouping these trivial packages to cope with the problems associated with publishing them. Then, to quantitatively investigate the impact of these trivial packages on the *npm* ecosystem and its users, we examine grouping these trivial packages. We found that if trivial packages that are always used together are grouped, the ecosystem can reduce the number of dependencies by approximately 13%. Our findings shed light on the impact of publishing trivial packages and show that ecosystems and developer communities need to rethink their publishing policies since it can negatively impact the developers and the entire ecosystem.

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

1 Introduction

Recently, Node.js/JavaScript has become one of the most commonly and widely used programming languages (StackOverflow 2020). One of the main forces behind the popularity of JavaScript is the fact that code sharing is prominent and even encouraged by its developer community (Cox 2019). This code sharing is supported by the node package manager (*npm*). For example, *npm* provides over 750,000 packages that developers can use.

Despite the advantages of such code sharing, there are some downsides of such code sharing that include lower software quality, increased maintenance effort and even legal issues (Abdalkareem et al. 2017; Zimmermann et al. 2019; Orsila et al. 2008). In an incident, the reuse of a very simple Node.js package called left-pad, which was used by another well-known package called Babel, caused interruptions to some of the largest Internet sites, including Facebook, Netflix, and Airbnb (MacDonald 2018). The dependency that caused Babel to break was merely a trivial 11-line package that implemented a left padding string function.

There have been studies that looked at why these trivial packages are used. In our prior work (Abdalkareem et al. 2017), we defined and examined trivial packages in the *npm* ecosystem, and discovered a number of relevant findings. we found that 1) trivial JavaScript packages tend to be small in size and less complex and they are prevalent, approximately making up 16.8% of all the packages on *npm*. 2) JavaScript developers generally use trivial packages since they believe that trivial packages provide them with well-tested code; however, they are concerned about the management of extra dependencies. Moreover, our study showed that in some cases, these trivial JavaScript packages can have their own dependencies, imposing significant overhead on the maintenance of software projects. However, this phenomenon has another side to the coin, which is why are these trivial packages even published in the first place. Is it for personal satisfaction, or is it for technical reasons? More importantly, what impact do publishing such trivial packages have on the *npm* ecosystem.

Therefore, we perform a study to understand why developers publish trivial packages, what they believe to be the drawbacks of publishing them, and how we can alleviate their negative effects if any. In particular, we examine the following research questions:

- ***RQ1****: What are the potential advantages of publishing trivial packages?* Prior work shows that trivial packages are commonly used in JavaScript projects and developers believe that they provide them with well-tested code (Abdalkareem et al. 2017; Abdalkareem 2017). However, the intention of publishing these trivial packages is still a question to be answered. In this research question, we aim to discover why developers publish trivial packages in the *npm* ecosystem in the first place.
- ***RQ2****: What are the potential disadvantages of publishing trivial packages?* Considering the fact that trivial packages exist and make up to approximately 16% of the *npm*

ecosystem, they bring with them many problems such as inflating the *npm* ecosystem. However, up-to-date, the impact of these trivial packages on the *npm* ecosystem has not been examined yet. Thus, in this question, we aim to investigate the negative impact of having such trivial packages.
- ***RQ3***: *How do developers believe that we can alleviate the overhead of having so many trivial packages?* Related to *RQ2*, we argue that JavaScript developers have dealt with the side effects of trivial packages. In addition, their experience related to these trivial packages can uncover some practical solution to alleviate the negative impact associated with these trivial packages. Therefore, we ask the question of how developers alleviate the drawbacks associated with these trivial packages.
- ***RQ4***: *What is the impact of grouping trivial packages on the npm ecosystem?* Based on the gained insight from the prior RQs, we found that developers provided many suggestions to potentially alleviate the disadvantages associated with trivial packages. But it is not clear how much (if any) these suggested alleviation strategies benefit the developers. Therefore, we set out to quantitatively examine the impact of the most cited alleviation strategy - grouping some trivial packages.

To perform our study, we started by analyzing the source code of more than 750,000 *npm* packages to identify trivial packages and the developers who publish them in the *npm* ecosystem. We found that out of all packages published on *npm*, 15.09% of them are identified as trivial packages. We also found that 28.7% of developers on *npm* has published at least one trivial package.

Once we identified developers who published trivial packages, we surveyed 59 JavaScript developers to answer our research questions. Our findings show that developers believe that *building reusable components*, *testing & documentation*, and *separation of concerns* are the main advantages for publishing trivial packages. However, developers reported several problems related to having these trivial packages in the *npm* ecosystem that include, the overhead of maintaining multiple packages, the increase of dependency hell, which means more complex dependency management for the *npm* ecosystem and for their projects as well, and difficulty in finding the right packages. Moreover, when asking developers about their strategies for alleviating the problems related to publishing trivial packages, we observed that grouping trivial packages, using dependency management tools, and providing better search tools are the most suggested actionable solutions by the JavaScript developers. These findings help the research and developer communities to have a more holistic view of the positives and negatives that trivial packages have on the *npm* ecosystem.

In addition, we investigated the applicability of applying the most suggested strategies from the developers to alleviate the problem of having so many trivial packages, which is *grouping these trivial packages*. We found that when trivial packages are grouped based on the co-usage, the *npm* ecosystem can reduce the number of dependencies by approximately 13%.

Finally, to assists the replication of our study, we made all the analyzed *npm* packages and survey responses publicly available (Chen et al. 2019).

Our work makes the following main contributions:

- We analyze more than 750,000 *npm* packages to identify trivial packages and the developers who publish them.
- We conducted a survey with 59 JavaScript developers to better understand why developers publish trivial packages and the problem associated with such practices. We also investigate how developers alleviate the reported drawbacks.

– We examine the practicality of applying the strategy of grouping trivial packages to eliminate their problems based on the developers' suggestions.

Paper Organization The remainder of this paper is organized as follows. The background and the motivation of our study presented in Section 2. In Section 3, we present our study design. In Section 4, we present the results of our empirical study, while we discuss some implication of our findings in Section 5. In Section 6, we discuss the related work. Section 7 discuses the threats to the validity of our analyses. Finally, Section 8 concludes our paper.

2 Background and Motivation

In our prior work (Abdalkareem et al. 2017), we examined why JavaScript application developers use trivial packages in the *npm* ecosystem. Specifically, we first started by defining what is a trivial package, which based on a user study with JavaScript developers, was defined to be packages that contained 35 lines of code or less and a McCabe's cyclomatic complexity less than or equal to 10. Using that definition, we mined the source code of 38,807 JavaScript applications to identify the developers who introduce and use at least one trivial package into the examined application. We then surveyed those developers who use the packages to understand the reasons for and drawbacks of using trivial packages. In total, we received responses from 88 JavaScript developers, and then we analyzed those responses.

Based on our study, we found that trivial packages are commonly and widely used in JavaScript applications, with around 11% of the analyzed JavaScript applications directly depending on at least one trivial package. As for the reasons why developers use trivial packages, we found that 54.6% of the developers in our survey believe that trivial packages provide them with a well-implemented and -tested code, and 47.7% of them stated that increased productivity. Developers also mentioned other reasons for using trivial packages that include improving the readability and performance of their code.

However, our findings show that developers (55.7%) believe that using trivial packages results in a dependency mess that makes it hard to update and maintain. Also, developers stated that using trivial packages makes their applications vulnerable to breakage and security issues. Moreover, when we empirical examined the most cited reason and drawback of using trivial packages, we found that contrary to JavaScript developers beliefs that trivial packages are well-tested, only 45.2% of trivial packages and even some of these trivial packages can bring more pain than they are worth since some have as many as 20 dependencies.

Although our prior work provided a comprehensive view of why developers use trivial packages, these findings raised the important question - why do developers even publish such trivial packages in the first place? Hence, complementary to our previous work in Abdalkareem et al. (2017), we want to know the real motivation that derives JavaScript developers to publish such packages even though our prior work showed some bad consequences of using them. For example, studying trivial packages from the publisher's perspective will reveal their impact on the JavaScript community and on the ecosystem that they belong to. For example, as we show later in this paper, while some developers believe that they publish trivial packages to build reusable components, others stated they publish trivial packages for their own personal satisfaction. Having such a perspective (i.e., the publisher's) helps us better understand the context in which these trivial packages are created and also helps the ecosystem understand how they might want to handle such packages.

3 Study Design

We would like to better understand the motivations of developers for publishing trivial packages and the impact of these packages on the *npm* ecosystem. Our study is a developer-centered and covers four main research questions:

– ***RQ1****: What are the potential advantages of publishing trivial packages?*
– ***RQ2****: What are the potential disadvantages of publishing trivial packages?*
– ***RQ3****: How do developers believe that we can alleviate the overhead of having so many trivial packages?*
– ***RQ4****: What is the impact of grouping trivial packages on the npm ecosystem?*

To answer the first three aforementioned questions, we conducted an online survey with JavaScript developers who publish trivial packages and qualitatively analyze their responses. To identify the potential survey candidate developers, we first analyzed the *npm* ecosystem, identified developers that actually publish trivial packages and sent out our online survey to them. To answer the fourth question, we analyzed the *npm* ecosystems and identify co-usage trivial packages. In the following subsections, we describe our data collection process and our survey design. We also describe our method of grouping co-usage trivial packages.

3.1 Trivial Package Dataset

Our study exclusively examines *npm*. We choose to examine the *npm* ecosystem for several reasons. First, *npm* is one of the largest and fastest growing software ecosystems - from 2017 to 2018, *npm* nearly doubled in size (DeBill 2019; Decan et al. 2018). Second, since we wanted to examine trivial packages, it is easier for us to build on prior work by Abdalkareem et al., who provided clear guidelines of how we can determine what a trivial package is (Abdalkareem et al. 2017; Abdalkareem 2017).

To obtain our dataset, we extracted the list of all published packages from the *npm* registry (npm Documentation 2020), which developers use to publish their packages, make them available for others to use. We collected the list of packages as of September 1st, 2018. We found that there are 752,012 registered *npm* packages. We extracted the `URL` of the tar file of all the packages. Then, we developed a crawler to clone them locally into our machines. Since some packages do not exist on the *npm* registry or are unpublished (i.e., we received a 404 error), we were not able to clone 4,298 of them. In the end, we cloned the source code of 747,714 *npm* packages published by 202,422 unique developers since some developers had published more than one *npm* packages.

Once, we had the 747,714 *npm* packages locally, we analyzed them to identify the trivial packages. To do so, we applied the definition provided by Abdalkareem et al. (2017), where they define trivial packages through conducting a user-study with JavaScript developers. In their user-study, Abdalkareem et al. asked developers to examine randomly selected *npm* packages and mark the ones that the developers consider them trivial packages. Then, they analyzed the marked packages based on their size and complexity and defined a trivial package as a JavaScript package that has $\left\{X_{LOC} \leq 35 \cap X_{Complexity} \leq 10\right\}$, where X_{LOC} represents the JavaScript LOC and $X_{Complexity}$ represents McCabe's cyclomatic complexity of the package X. We refer readers to the original paper by Abdalkareem et al. (2017) for a full detailed explanation of the user study.

To apply this definition to all the packages in our dataset, we use the Understand tool by SciTools (Tool 2020). Understand is a source code analysis tool that provides various code metrics. Following our analysis, we found that there are 112,833 trivial packages in our dataset, which represents 15.09% of all *npm* packages.

To better understand our dataset, we perform some basic analysis. We perform this analysis to extract information about the developers who published trivial packages. We found that the 112,833 trivial packages in our dataset are published by 58,012 developers, which means that 28.7% of developers on *npm* have published at least one trivial package. Table 1 shows various statistics for developers on *npm* based on their published packages–trivial and all.

We observe that overall, the mean number of packages published by developers on *npm* is 3.73, while the mean for publishing trivial packages is 0.56 for all the developers on *npm*. However, when we examine only the number of packages published by developers who publish at least one trivial package, we observe that the mean is 7.13. The table also shows that on average developers who published at least one trivial package had published 1.94 trivial packages. This analysis shows that developers who publish trivial packages are productive and active developers within the *npm* ecosystem.

3.2 Developer Survey Design

We designed an online survey that included five questions. We first asked two questions about the participants' background. Then, we asked developers three open-ended questions to give respondents maximum flexibility to express their opinions, and these questions were optional. These questions were 1) what they believe to be the main advantages of publishing trivial packages, 2) what they believe to be the disadvantages of publishing trivial packages, and 3) how they would alleviate the disadvantages of publishing trivial packages.

Our dataset contained 202,422 JavaScript developers. Since this number is quite large and most of them publish only one package, we decided to only survey developers who had published at least 10 trivial packages on *npm*. Also, these selected developers are considered to be active developers and have an experience of publishing trivial packages since they published at least 10 trivial packages. We found 876 developers who met this condition. We then randomly selected 250 developers among these 876 developers and collected their name and emails. We selected this number since we did not want to overwhelm the *npm* developer community with our invitation to the survey. The invitation email included the number of detected trivial packages published by the developer and named one of the published trivial packages as an example. We received 59 responses in the first three days after making the survey available online for a week (i.e., the response rate is 23.6%). We believe that this response rate is acceptable and within the line with other survey-based studies in software engineering (Singer et al. 2008).

Table 2 shows the developers' experience and their positions. The Table shows that of the 59 participants in our survey, 47 have more than five years of development experience, 7 had between four to five years of development experience, and only 5 developers have less than three years of development experience. As for the participant's positions, the majority of them are working in industry with a number of 43 and 7 of them are freelance developers. The Table also shows that 9 developers have other backgrounds like researchers or company owners. Overall, the participants' background in our survey shows that most of them are experienced developers.

Table 1 Summary of the number of packages published by developers on *npm*, the number of packages (trivial and all) for all developers, and for only developers who publish at least one trivial package

Packages	All developers who published packages				Developers who published trivial packages			
	Min.	Mean	Median	Max.	Min.	Mean	Median	Max.
All Packages	1.00	3.73	1.00	4,711	1.00	7.13	2.00	4,711
Trivial Packages	0.00	0.56	0.00	1,340	1.00	1.94	1.00	1,340

Table 2 Background of survey participants

Experience	#	Developers' Position	#
1 - 3	5	Developer working in industry	43
4 - 5	7	Freelance Developer	7
>5	47	Other	9

3.3 Manual Analysis of Survey Responses

Since our study is qualitative in nature, and our survey questions are open-ended free-form text, we performed a formal process to analyze the results from our survey (Seaman 1999). For each of the open-ended questions, we extract the participants' responses and print them. Then, we conduct a three-stage process. In the first step, the first two authors carefully read the participants' answers and came up with an initial list of categories that the responses fell under. After that, the two authors discussed these initial categories to come up with the final themes. In the third step, the same two authors went through the responses and classified them according to the extracted themes. To confirm that the two authors correctly classified the responses to the right category, we measure the classification agreement between the two authors. We use Cohen's Kappa coefficient (Fleiss et al. 2013). Cohen's Kappa coefficient is a widely used statistical method that evaluates the inter-rater agreement level for categorical scales. This coefficient has a range between -1.0 and 1.0, where -1 means negative agreement, 0 indicates no agreement, whereas 1 is full agreement. In our work, we consider the value of this coefficient to be excellent inter-rater agreement when it is bigger than 0.75 (Fleiss et al. 2013). Finally, for the responses that the two authors failed to agree on, the third author was consulted to resolve the differences and help to categorize these responses.

3.4 Grouping Co-usage Trivial Packages

To quantitatively examine the impact of the most cited alleviation strategy suggested by our survey participants, we examine the idea of grouping some trivial packages.

One of the first questions that arises is what strategy should be used to perform this grouping? One of the most intuitive ways to group trivial packages is based on their co-usage. The idea is that if two or more trivial packages are frequently used together in a software application, then it would make sense to group these trivial packages together. For example, given two packages, $Pack_x$ and $Pack_y$, we measure their co-usage as a ratio of their average occurrence. Here, the average is simply the occurrence of the individual packages divided by the size of the group. For instance, if $Pack_x$ and $Pack_y$ have been used 10 and 9 times, respectively, and their co-usage is 9 (i.e., they appeared together 9 times), then the frequency of co-usage is given as $\frac{9}{\frac{10+9}{2}} = 0.95$. This frequency of co-usage of a group of packages can range between 0 and 1; where a high value indicates that the co-usage is popular.

For the sake of our investigation, we decide to group trivial packages where the majority of their appearances are together, i.e., we group any packages that have a co-usage $\geq$ 0.5. Also, due to the extensive computation power required to run our analysis, we count only the co-usage groups that appear more than forty times in our dataset.

We extracted the co-usage of trivial packages in the entire *npm* ecosystem. To do that, we parsed the `packages.json` of the 747,714 packages in *npm*. Given the sheer volume of packages and possible combinations, we limited our analysis to co-usage groups that had at least 40 appearances. In total, we found 87,092 cases of trivial package co-usages in our dataset.

4 Study Results

In this section, we present our study results that help us better understand why developers publish trivial packages, what they believe to be the main disadvantages of publishing trivial packages, if any and how do they alleviate these drawbacks.

4.1 *RQ1:* What are the Potential Advantages of Publishing Trivial Packages?

To understand why developers publish trivial packages, we explicitly ask survey participants about the main advantage(s) for publishing trivial packages. After our manual analysis of the survey responses, we were able to extract seven main reasons for publishing trivial packages. The 'Other' class was added to classify reasons that rarely appeared and/or did not fit into any of the major reasons. We also examine the level of agreement between the two annotators and found that the value of Cohen's Kappa coefficient is equal to 0.84, which is considered to be an excellent agreement (Fleiss et al. 2013).

Table 3 shows the extracted reasons and the percentages of responses that correspond to each reason. Since some developers provide more than one reason for publishing trivial packages, the percentages may add up to more than 100%. We detail each reason below:

1. **Building Reusable Components (64.41%).** The most frequently cited reason for publishing trivial packages is the creation of reusable components. Even though these trivial packages are typically small, and in most cases can be easily written by the developers themselves, developers believe that creating a reusable component out of them is of value. For example, P4, P17, & P19 stated that P4 *"I can compose complex application using 'lego blocks' of npm modules, easily swapping on out for another when necessary"*, P17 said that *"Avoid re-implementing the same things across projects."* & P19 *"Publishing a package that does one thing well means never having to write that*

Table 3 Reasons for publishing trivial packages

ID	Reasons	Responses (%)
1.	Building Reuse Component	64.41%
2.	Tested and Document	33.90%
3.	Separation of Concerns	32.20%
4.	Optimization	27.12%
5.	Helping the Community & Personal Satisfaction	22.03%
6.	Maintenance Overhead	16.95%
7.	Dependency Management	6.78%
8.	Other	11.86%

Responses can sum to more than 100% since respondents can report more than one reasons *code again."* Although this reason seems to be obvious, it is interesting to see developers taking the benefit of code reuse to the extreme where they create a package even for such simple tasks.

2. **Test and Documentation (33.90%).** The second most cited reason by developers for publishing trivial packages is the fact that trivial packages can be easily tested and documented. Approximately 34% of the developers in our survey mentioned that because trivial packages are small, they can be easily tested in isolation. For example, P50 stated that *"They are easier to write tests for because they are small."* and P27 said *"Another advantage of small modules is that when a developer chooses some subset of their current problem, they can focus on handling every edge-case and error in which the function can be used"*. The developers also believe that providing documentation for trivial packages is easy since they only provide simple tasks. For example, P18 explains this as follows *"Most of the very small packages are actually more mean as documentation, e.g., 'this is how I solved it' "*. The provided responses may be indicating a bigger challenge, which is the lack of tools and techniques to effectively test JavaScript applications (Fard and Mesbah 2017). Either way, it is interesting to see that developer consciously publish these small packages since they ease testing and documentation.
3. **Separation of Concerns (32.2%).** Developers also reported publishing trivial packages help them to focus on the implementation of specific tasks. More than 32% of the developers mentioned that they publish trivial packages because these packages implement a very specific task. Trivial packages are small, hence, they can be combined into the applications more easily and help to keep the applications code small and clear. For example, developer P37 said *"The main advantage is isolating the problem down to a scope of a tiny re-usable component"*, developer P30 said *"Separation of concerns"*, and P35 said *"Base of common solutions to common [micro] problems. I would say micro packages are always better than heavy all-in-ones"*. Interestingly, some developers link the publishing of trivial packages to one of unix's philosophy of writing small programs (Wikipedia 2018), as one developer P57 stated *"It fits the unix model of small sharp tools that do one thing well and can be composed to do complex things."*
4. **Optimization (27.12%).** Developers also reported that creating trivial packages can provide them with 'lightweight' packages. Compared to larger packages, trivial packages require less space with zero dependencies since they provide one functionality. One developer summarized these advantages as follows: P15 said *"They 'trivial packages' don't require the same amount of coordination as large framework or library, are easier to compose and take up less space in browser bundles."* Another developer P43 stated that *"it has a small size when installed so opening a web app or downloading a mobile app is faster."*
5. **Helping the Community & Personal Satisfaction (22.03%).** Since JavaScript developers often rely on external packages due to the lack of a comprehensive standard JavaScript library (Abdalkareem et al. 2017; Fuchs 2016), developers reported that they publish trivial packages to provide other developers with useful code. For example, P2 & P32 said *"I use npm as snippet library with docs/test with the added benefit of the community."* & *"Finishing those pieces becomes obviously useful and important, because you know that people outside of your immediate team will need them."*

   In addition, to the reasons of helping the community, four developers in our survey said that they get personal satisfaction when they contribute, especially when their packages are used by other developers. For instance, the participant P28 wrote *"satisfaction from seeing people downloading my packages, [...], the feeling of helping* *others"*. This is a very interesting observation which shows that in some cases, there are non-technical reasons for publishing these trivial packages.

6. **Maintenance Overhead (16.95%).** Several developers consider the saving of maintenance effort through published trivial packages. The fact that these trivial packages are small with simple functionality makes them require less updates, unless they have significant changes. This reason was clearly described by developer P4 who said *"Less maintenance cost, a module with a very tight scope and simple set of features (with no plans to expand to a broader scope) will likely not require much updates or maintenance by the author."* & P50 said that *"They are also easier to version because there is less code to change and so a version update is almost always significant to the entire package"*. There is of course a flip side to this, since having too many packages to keep track of may introduce maintenance overhead of another kind. We discuss this in more details later in the paper.
7. **Dependency Management (6.68%).** Developers also report the fact that publishing trivial packages can provide developers with easy to manage dependencies/packages. For example, developers believe that trivial packages are much easier to manage than large JavaScript framework e.g., P51 *"If your dependencies are small, it's easier to swap them out for other code. Once you start depending on 'swiss army knife' style modules, it becomes harder to migrate away."*

Summary of findings: JavaScript developers reported that they publish trivial packages for sharing reusable components, testing and documentation, separation of concern, and for optimization reasons.

4.2 *RQ2*: What are the Potential Disadvantages of Publishing Trivial Packages?

On the flip side, we also wanted to examine whether developers see any potential disadvantages of publishing trivial packages. To extract the main disadvantages of publishing trivial packages, we followed the same process used to analyze the answers of the first RQ. We

Table 4 Drawback for publishing trivial packages

ID	Problem	Responses (%)
1.	Maintaining Multiple Packages	35.59%
2.	Dependency Hell	22.03%
3.	Find the Right Packages	15.25%
4.	Duplicate Packages	13.56%
5.	Long Time to Install	8.47%
6.	Other	22.03%

Responses can sum to more than 100% since respondents can report more than one problem also found the inter-rater agreement between the two authors to be 0.86, which is considered to be an excellent agreement (Fleiss et al. 2013).

Table 4 lists the extracted disadvantages and the percentage associated with each stated disadvantage. We detail each specified disadvantage below:

1. **Maintaining Multiple Packages (35.59%).** The most reported disadvantage is the fact that publishing trivial packages can increase the work-load for developers to maintain and keep track of each package. For example. P2 said that *“Additional tooling for maintaining lots of repo and modules”*. P8 *“Maintaining packages on npm is annoying. Also, github alerts/issues does’t scale well across many repositories”*. Also, developer P40 stated as a disadvantage that *“Keeping track of everything I have published.”* This is interesting in light of the fact that one of the advantages of publishing trivial packages is lower maintenance. However, we believe that publishing many trivial packages introduces a different type of maintenance, which involves keeping up with many different packages.
2. **Dependency Hell (22.03%).** Developers also had trouble dealing with multiple dependencies that trivial packages may cause. For instance, developer P16 reported that *“For me, the disadvantage is when you are several levels deep and need to push a patch. If you patch one package that two other levels are relying on, those two levels then should update the dependency and publish new versions as well. This can be a pain, but ultimately is much better of a plan than copy/pasting the code or not being specific with semver, which both cause enormous and horrible problems.”* While this issue can be directly related to the application level, developers also reported this concerning on the *npm* ecosystem level. For example, P18 stated *“The ecosystem gets larger dependency trees. With many developers and packages that can break at any time.”*
3. **Finding the Right Packages (15.25%).** Another problem reported by the developers is the lack of an advanced searching tool, especially when there are many packages that do similar things. Developers believe that publishing trivial packages increases the size of the ecosystems. For example, P14 & P1 explain this concern. P14 said *“There are too many alternative for everything, is not always clear what is the benefit for each one.”* and P1 said *“The number of quality packages that match your search query might be significantly less then the number of total packages”*. This point is very important, since *npm* as an ecosystem must deal with such undesired evolution or it becomes too noisy to be effective for developers.
4. **Duplicate Packages (13.56%).** JavaScript developers also reported how publishing trivial packages can lead to having duplicated packages in the ecosystem. For example, P42 said *“Small packages are remarkably easy to write and publish and therefore make it easy to ‘pollute’ the ecosystem with duplicated entries.”*, P7 said *“Many repeated packages”*, and P27 *“They are sometimes replicated, or are replicated with small changes.”* This findings is related to the previous disadvantage since having duplicates also makes it harder to find the right package.
5. **Long Time to Install (8.47%).** The least cited disadvantage reported by developers is the fact that publishing trivial packages can cause long installation times. In general, trivial packages provide one single functionality, and in most cases developers depend on several of these trivial packages, which as a results increases the number of dependencies of a project. Thus, installing such a long list of dependencies may result in long installation and build times. For instance, P6 stated that *“As an end user I often end up with tens of thousands of packages installed in my projects and it takes a lot of time to install them.”*

Other category was added to categorize disadvantages that are rarely reported and/or did not fit into any of the major categories that we extracted.

Summary of findings: Maintaining multiple packages, dependency hell, and finding the right packages are the most reported disadvantages of publishing trivial packages.

4.3 *RQ3*: How do Developers Believe that we can Alleviate the Overhead of Having so many Trivial Packages?

In addition to discovering the disadvantages of publishing trivial packages, we also asked the developers how they would alleviate the issues related to the disadvantages of publishing so many trivial packages. Of all the survey respondents, 88.14% of them shared their solutions on how to alleviate the disadvantages they mentioned in the previous survey question. Again, since the responses of this survey question is free-text, we perform a formal manual analysis and measure level of agreement between the two annotators. We found the inter-rater agreement to be 0.79.

Table 5 shows the extracted alleviation strategies suggested by developers. We detail each alleviation strategy below:

1. **Grouping Packages (25.00%).** The most commonly cited alleviation is to group multiple trivial packages. In fact, developers blame publishing trivial packages on many issues, which include maintaining multiple packages, dependency hell, etc. For example, developer P46 explicitly stated that *"Combine multiple small packages into a single library for an application archetype"*. Also, developers reported solution of hosting trivial packages that they proposed or already used. This strategy is well explained by P2 & P38 who said P2: *"Better multiple-packages repositories (monorepo) tooling"* & P38: *"You can put several packages in one repo"*. Although this is suggested by developers, how to group the trivial packages is a different issue that we plan to work on next. As we show later, applying this strategy can save enormous resources.
2. **Dependency Management Tools (11.54%).** Developers also suggested managing dependencies through better tools. As the previous question showed, more than 22%

Table 5 Alleviation for publishing trivial packages

ID	Alleviation Strategies	Responses (%)
1.	Grouping Packages	25.00%
2.	Dependency Management Tools	11.54%
3.	Better Search Tool	7.69%
4.	Writing Better Tests & Documentations	5.77%
5.	Improving the Ecosystem Policy	3.85%
6.	Removing Duplicate Packages	3.85%

of the developers are concerned about the extra dependencies. Even though this issue is a well-studied problem in the literature (Mirhosseini and Parnin 2017), developers still face these issues and need better solutions. Developers reported having dependency management tools as a possible alleviation to this problem. For instance, developers P30 stated *"A tool like greenkeeper makes it possible."* and P16 *"Services like greenkeeper and renovate help to update dependencies easily. You can even set them to run your tests, and if they pass, auto-merge the PR. This way the only thing left up to the author is publishing the package."*

3. **Better Search Tools (7.69%).** Related to the problem of finding the right packages, developers are interested to have a better search engine. In fact, there have been several attempts to build search functionality into the *npm* ecosystem e.g., the "npms",[1] "npm discover tool",[2] and the "npm packages PageRank" tool.[3] Most of these search engines use the keywords provided by package developers and other popularity measure, such as the number of downloads and starts to rank packages. Developers believe that more advanced search engines are need, e.g., search engines that use the semantics of the source code. For example, developer P59 explains this idea in their response: *"Make a better search engine for packages, somehow they should look at the actual code as well so that you could find modules having certain code snippets no matter what they are named."*
4. **Writing Better Tests & Documentations (5.77%).** More than 5% of the developers see that several problems related to publishing trivial packages, namely dependency hell and finding the right packages can be simply mitigated with better tests and documentation. For instance developers P7 & P40 stated that P7: *"write test for a npm package, this can help me to give up publishing useless packages."*. P40: *"be more diligent in what i publish & actually writing documentation"*.
5. **Improving the Ecosystem Policy (3.85%).** A less commonly proposed solution by developers is to improve or change the software ecosystem's policies. For example, P35 stated *"Better npm policy regarding package quality: some base requirements, like tests, correspondance of name and function, responsibility of an author, reasonable package size/dependencies, shared control over the package."*. In fact, software ecosystems continue to improve their regulation and policies. A clear example of such policy changes is the incident of left-pad where *npm* maintainers changed the policy to not allow developers to unpublish packages from *npm*, which was the main cause of the incident.
6. **Removing Duplicate Packages (3.85%).** Other JavaScript developers in our survey propose the idea of removing duplicated packages from *npm*. Developer P27 explain this idea. *"Multiple modules which do the same thing can be consolidated, and the old ones deprecated and have their usages replaced in public repositories."*.

Also, developers reported some other less common recommended solutions to alleviate the disadvantage of publishing trivial packages, which do not really warrant their own category. For example, P4 *"Better GitHub integration with npm, for example the*

[1] https://npms.io/
[2] http://www.npmdiscover.com/
[3] http://anvaka.github.io/npmrank/online/

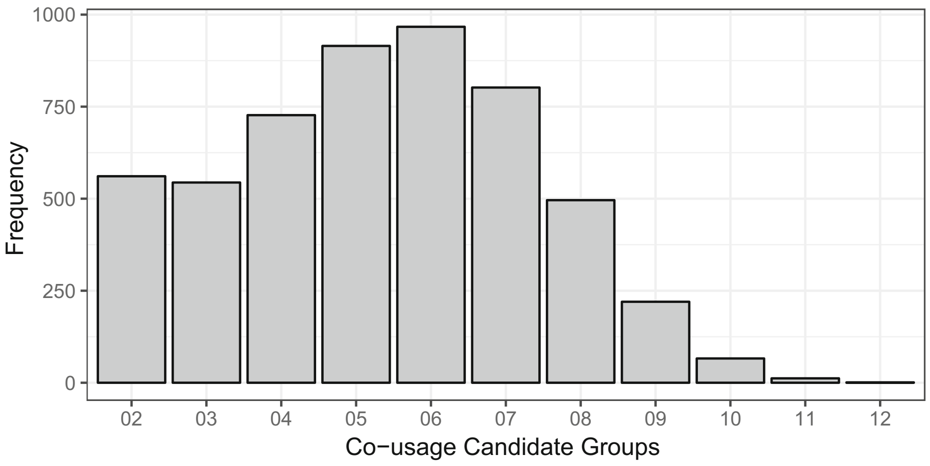


Fig. 1 The bar-plot shows the distribution of co-usage candidate group of trivial packages

ability to publish a patch/update to npm from GitHub would allow me to update modules much more quickly.” & P50 *“Provide more information to the community about why relying on trivial/small packages is good.”*

Summary of findings: Developers proposed six strategies to alleviate the disadvantages of having trivial packages on the ecosystem. The most popular strategy is to combine trivial packages into one large package.

4.4 RQ4: What is the Impact of Grouping Trivial Packages on the *npm* Ecosystem?

We examine the most reported alleviation of publishing trivial packages by our survey participants, which grouping co-usage packages. To do so, we analyzed the entire *npm* ecosystem and identify co-usage trivial packages.

Figure 1 shows bar plots of the co-usage trivial package groups (frequency vs. group size). We find 3,762 groups from the 87,092 cases, with group sizes ranging between 2-12 trivial packages. The Figure shows that the most frequent co-usage trivial package groups sizes are five, six, and seven, with the number of group being 915, 967, and 802, respectively. Even though these co-usage group are quite large in some cases (i.e., they contain more than five trivial packages in one group), this figure clearly illustrates that these trivial packages can be grouped together.

In some cases, even the package names are clear indicators that they should be grouped somehow. For example, one group containing six trivial packages, that has a frequency co-usage value of 0.8 has the following packages in it [`validate.io-number-primitive`, `validate.io-matrix-like`, `validate.io-array-like`, `validate.io-nan`, `validate.io-typed-array-like`, `validate.io-positive-primitive`]. Given that all these packages start with validate.io, it is almost clear that they should be grouped, yet they are not.

After determining the potential groups of trivial packages, we would like to examine the impact of this grouping. To do so, we examine the direct impact of this grouping in

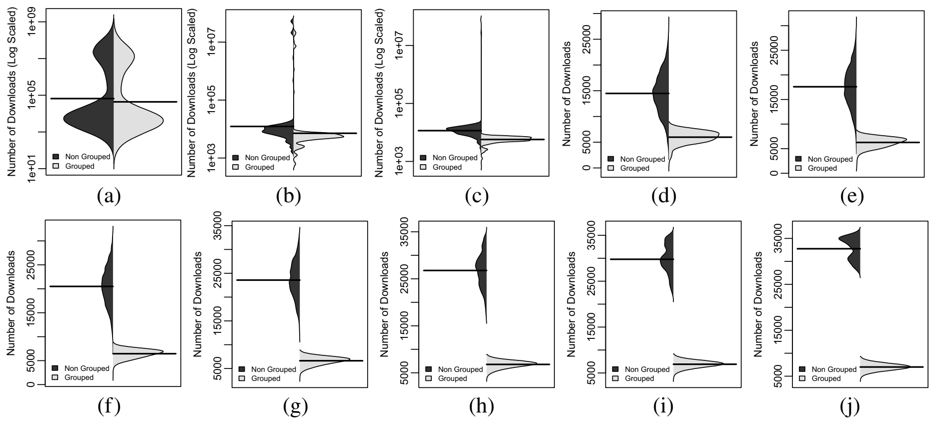


Fig. 2 Beanplots showing the distributions of the number of download of non grouped and grouped trivial packages based on their group size. The horizontal lines represent the medians

two complementary ways. We measure both the number of downloads and the number of dependencies that we can save by grouping these trivial packages. Hence, we first compare the downloads for the trivial packages in the case that they are grouped (based on co-usage) or not (which is what is happening now).

Figure 2 shows bean-plots for the download count for non-grouped vs. grouped trivial packages. We can see that as expected in most cases, the download count is lower in the case where trivial packages are grouped. The download count for a group of packages is taken as the maximum number of downloads for any package in the group. For example if a group has three packages that are downloaded 10, 20 and 50 times, then we would take 50 as the group download count. Figure 2 shows the average download count for non-grouped trivial packages based on the group size ranges between 7,031,470 (group of two packages) and 32,756 (group of eleven packages) while it ranges between 3,979,235 and 6,997 for the trivial packages after they are grouped. We perform a Mann Whitney test (Mann and Whitney 1947) to determine if the difference between the two distributions is statistically significant (with a p-value = 0.05). We find that the difference is statistically significant in all cases, expect for the group size of two trivial packages.

Second, we also quantitatively examine the percentage of the saved dependencies in the *npm* ecosystem when the co-used trivial packages are grouped. To do so, we count the total number of dependencies *npm* packages have before grouping the trivial packages and after grouping them. As we describe earlier, we found 3,762 different group of co-used trivial packages that we count their dependencies. We found that the total number of dependencies before the grouping is 16,166 and found number of dependencies after grouping the co-used trivial packages equal to 14,074. Then, we calculate percentage of the saved dependencies in the *npm* ecosystems after grouping the trivial packages as following: $\frac{16,166-14,074}{16,166} * 100 = 12.94$.

This analysis confirms developers' belief about the disadvantages of publishing trivial packages in the *npm* ecosystems, for example, the dependency overhead problem. It also shows that suggested alleviations to reduce the drawback of having published trivial packages are useful. In particular, our results show that approximately 13% of the dependencies can be saved in the *npm* ecosystem by grouping co-used trivial packages.

Summary of findings: Our results show that grouping co-usage trivial packages can save approximately 13% of the number of dependencies in the npm ecosystem.

5 Implications

In this section, we discuss the potential implications of our findings for both practitioners and researchers.

5.1 Implications for Practitioners

Our findings show that publishing trivial packages introduces several problems that may impact other users and the entire *npm* ecosystem. The first direct implication of our results is that developers should rethink their publishing policies. Developers should ask several systematic questions before deciding to publish a trivial package since such packages do come with an associated cost (i.e., maintenance of these packages, support of users of these packages, etc.). Some questions that developers should ask include: Is the newly published trivial package of good quality? Does the new package warrant its independent publication or is it better added it to an existing published package? Will I have the resources to appropriately maintain this as a new package (e.g., releasing patches)? In addition, our findings reveal that having too many trivial packages may introduce "noise" or inflate the ecosystem. This leads to extra work for the maintainers of the ecosystem itself who need to periodically perform tasks to maintain the health of the ecosystem. Such tasks include removing duplicate packages, scanning packages for vulnerabilities, and facilitating effective search of relevant packages. Thus, developers should not assume that publishing a trivial package is a free lunch. We recommend that developers perform a careful investigation before publishing packages to make sure such packages are actually needed.

The second direct implication is for the *npm* ecosystem maintainer. *npm* maintainers should consider introducing a software quality practice to make sure that newly published packages are not negatively impacting the ecosystem. For example developer, P35 indicated that *"Better npm policy regarding package quality: some base requirements, like test."* Other interesting mechanisms that can be implemented by the *npm* maintainers suggested by the developer P14, who stated that *"Detect which of them are not being used and remove them after being some days in quarantine"*. We believe that putting these recommendations in place will increase the benefits of publishing trivial packages while mitigating their negative impact. Even if such a check is not possible, publishing some official guidelines to developers stating the minimum utility of a package or a policy to avoid duplication would be helpful for the ecosystem as a whole.

5.2 Implications for Researchers

While the findings of our study show that developers reported several advantages of publishing trivial packages, developers also reported some disadvantages of having trivial packages and provide some alleviations of these disadvantages. We believe these reported disadvantages and alleviation open a wide range of new research opportunities.

First, our study exposes the need for appropriate searching tools since developers reveal that publishing trivial packages increases the number of packages in the ecosystems, making it difficult to find the right package. While there exist some tools e.g., npms,[4] future research needs to be conducted experiments to evaluate these tools and make improvements to these exciting tools. One example is the need to go beyond popularity metrics such as downloads when ranking packages and considering other semantic- or functionality-related attributes when ranking searched packages.

Second, developers reported that publishing trivial packages leads to an increase in the number of duplicated packages in *npm* ecosystem. We believe that empirical studies that examine how prevalent such duplicated packages really are and how to alleviate the impact of such duplication are needed to support the ecosystem community. Although there have been some studies that examine duplicated repositories on GitHub (e.g. Lopes et al. 2017; Gharehyazie et al. 2017), we are not aware of any study that investigates or proposes techniques to detect duplicate ecosystem packages. In addition, survey respondents also pointed out that improving the *npm* ecosystem policy will alleviate some of the drawbacks of publishing trivial packages. Our findings motivate future work to empirically evaluate the current *npm* ecosystem policies in order to guide new contributions (or other) policies. We feel that this is an important area that needs much work. Finally, to examine the grouping strategy of trivial packages, we performed an analysis based on the co-usage. One future work is to build a tool that automatically combines these co-usage trivial packages and empirically examines its usefulness for both JavaScript developers and the *npm* ecosystem.

6 Related Work

In recent years, analyzing the characteristics of software ecosystems has gained momentum (Jansen et al. 2013; Serebrenik and Mens 2015; Sawant et al. 2018; Vasilescu et al. 2016; Trockman et al. 2018; Valiev et al. 2018; Linares-Vásquez et al. 2014a; Aghajani et al. 2018; Bavota et al. 2015; Abdalkareem et al. 2020). In this section, we discuss the work that are related to our study.

As mentioned earlier, in our prior work (Abdalkareem et al. 2017), we examined the reasons and drawback of using trivial package from the prospective of JavaScript users. In that study, we mined approximately 230,000 *npm* packages and 38,000 JavaScript projects to examine why JavaScript developers resort to use trivial packages in their software projects. We found that trivial packages are common and developers believe that they offer them with well-tested code. However, our analysis also showed that some of these trivial packages can bring more pain than they are worth, since some have as many as 20 dependencies.

Recently, Decan et al. (2018) investigated the evolution of packages dependency networks for seven software ecosystems. Their findings revealed that the studied software ecosystems grew over time in terms of number of published and updated packages. They also observed an increase in the number of transitive dependencies for 50% of the studied packages. Wittern et al. (2016) investigated the evolution of the Node Package Manager (*npm*) ecosystem in an extensive study that covered the dependence between *npm* packages, download metrics, and the usage of *npm* packages in real JavaScript applications. One of

[4] https://npms.io/

their main findings was that *npm* packages and updates of these packages are steadily growing. Also, more than 80% of packages have at least one direct dependency package that results in complex dependency network of the ecosystem.

Other areas of research focus on investigating the challenges of reusing packages from software ecosystems. Bogart et al. (2016) empirically studied three software ecosystems–Eclipse, R/CRAN, and Node.js/*npm*. They found that developers struggle with changing versions of the packages as the changes might break dependent code. They also found that developers demand to have techniques to identify breaking changes in upstream packages, especially those that are not correctly signaled by semantic versioning. Bavota et al. (2015) studied the evolution of dependencies in the Apache ecosystem and highlighted that dependencies have an exponential growth and must be taken care of by developers. Considering that changes of a package might break its dependent packages, they found that developers were reluctant to upgrade the packages they depend on.

Interestingly, previous research by Abdalkareem et al. (2017) and Bogart et al. (2016) found that although *npm* packages make it easier for developers to publish and use packages, *npm* does not review or test the packages, thus, the users decide which packages suit their needs better, leading to a healthy competition between similar packages in terms of functionalities in *npm* ecosystem.

Kula et al. (2018) studied the update of reused packages from software ecosystems and the types of these updates. They found that developers tend not to update their dependencies even though these updates are related to the addition of new features and patches to fix vulnerabilities. Scholtz et al. (2018) stated that outdated dependencies might cause security vulnerabilities, and they developed a tool that can automatically detect dependencies security risks. The tool can also update the outdated dependencies by running the test of the projects dynamically to check whether an update will break the code.

Sawant et al. (2018) found that APIs are a tremendous resource when they are stable, but the API deprecations also have impact on the projects that use these API. Such dependency impact is also observed in our study. Linares-Vásquez et al. (2014b) found that developers use social media to learn about the changes of software ecosystems, showing their concern about the impact of the changes.

Compared to the prior work that studies software ecosystems by focusing on the overall structure of an ecosystem, our study takes a deeper step with an analysis of the ecosystems from an inside view. Besides gaining a deeper understanding of the impact of publishing trivial packages on the *npm* ecosystem, we examine investigate strategies proposed the JavaScript developers to mitigate the problems of publishing trivial packages in *npm*.

7 Threats to Validity

We discuss the limitations of our study and the applicability of the results derived through our approach. For this purpose we discuss our work along three types of validity (Yin 2009) that include internal validity, construct validity, and external validity.

7.1 Internal Validity

Internal validity concerns factors that could have influenced our analysis and findings. First, in our study, we rely on the definition of trivial packages that is provided in prior work (Abdalkareem et al. 2017) to determine trivial *npm* packages. Although this definition of trivial packages was established based on surveying JavaScript developers, it maybe there are other possible definition for trivial packages. However, none of the 59 survey participants that we emailed about publishing trivial packages indicated that they are not trivial packages. This is a support that the used definition of trivial packages applies in the vast majority of cases. Second, to identify trivial packages in the *npm* ecosystem, we use the Understand tool to measure the number of line of code and cyclomatic complexity of the trivial packages. Hence, we heavily rely on the accuracy of Understand tool in extracting the studied measurements. However, the extensive use of Understand tool gives confidence in its analyses and results. We also use the definition of trivial packages provided by Abdalkareem et al. (2017). However, this definition, consider only analyzing JavaScript code and its variation such as Type Script. However, *npm* packages may contain code from other programming languages such as C/C++. To mitigate this threat, we examine a sample of the identified trivial packages, and we found that all examined cases that flagged as trivial packages are correctly classified. Finally, to count unique developers of trivial packages, we extracted the names and emails of the developers from the *npm* registry, which may introduce the threat of counting duplicated developers. However, *npm* enforces developers to have a unique identity in the registry, which gives us confident of using this approach.

7.2 Construct Validity

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. The analysis of responses provided by participants was performed manually, thus giving rise to potentially subjective judgment. To mitigate this threat, we employ a formal analysis and measure the agreement between the first and second authors. We found that the level of agreement between the two authors to range between +0.79 and +0.86, which shows excellent agreement levels. To answer our research questions, we conducted an online survey. We received 59 responses emails out 250 that we sent, which translate to 23.6% responses rate. While our response rate may be considered small, it is comparable or even higher than response rates reported in other software engineering surveys (Singer et al. 2008). Also, self-selection bias may influence our study since we only survey developers who publish trivial packages. To mitigate this bias, we select the developers randomly who publish different number of packages (trivial and non-trivial).

7.3 External Validity

Threats to external validity concern the generalization of our findings. In this study, we focus on one specific software ecosystem, which is the *npm* that is one of many software ecosystems. Hence, our finding and observation may not generalize to other software ecosystems. Also, the software ecosystem that we studied is the main software ecosystem for JavaScript programming languages. Thus, our results may not be generalized to other programming languages. That said, *npm* is one of the largest software ecosystem, which gives us confidence in our findings. In addition, to ensure that our survey participants understand what constitutes a trivial package, we surveyed 59 JavaScript developers who had published at least ten trivial packages in the past. Surveying a larger number or a different population of developers may lead to different results. Finally, we only surveyed developers who publish at least ten trivial packages since they have experience in dealing with and alleviating problems related to publishing trivial packages. That said, our findings may not be generalized to other developers who never publish trivial packages.

8 Conclusion

In this paper, we perform an empirical study to gain a better understand of the phenomenon of publishing trivial packages on *npm*. In particular, we answer questions that include why developers publish trivial packages, what are the drawbacks of publishing trivial packages, and how do developers alleviate the problems relate to publishing such trivial packages.

To answer these questions, we analyzed more than 750,000 packages that are published on *npm* to identify trivial packages and their authors. We then conducted a survey with 59 JavaScript developers who published at least one trivial package in order to answer our research questions. Our findings show that developers tend to publish trivial packages for the reasons that they help them building reusable components, testing & documentation, separation of concern. Even the developers who publish these trivial packages reveal that publishing such packages has some problems that include the maintenance of multiple packages, dependency hell, finding the right package, and increase the duplicated packages in the *npm* ecosystems. Moreover, we found that the majority of the developers in our survey suggested that trivial packages can be grouped together to alleviate problems associated with publishing them. Then, to quantitatively examine the impact of these trivial on the ecosystem itself. We found that if trivial packages that are always used together, and grouping them will help the ecosystem to reduce the number of dependencies by approximately 13%.

Acknowledgments We thank the JavaScript developer community and *npm* developers and special thanks for the developers who kindly respond to our survey.

References

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 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017. ACM, pp 385–395

Abdalkareem R, Oda V, Mujahid S, Shihab E (2020) On the impact of using trivial packages: an empirical case study on npm and pypi. Empir Softw Eng 25(2):1168–1204

Abdalkareem R, Shihab E, Rilling J (2017) On code reuse from stackoverflow. Inf Softw Technol 88(C):148–158

Aghajani E, Nagy C, Bavota G, Lanza M (2018) A large-scale empirical study on linguistic antipatterns affecting apis. In: 2018 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 25–35

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015) How the apache community upgrades dependencies: an evolutionary study. Empir Softw Eng 20(5):1275–1317

Bavota G, Linares-Vásquez M, Bernal-Cárdenas CE, Penta MD, Oliveto R, Poshyvanyk D (2015) The impact of api change- and fault-proneness on the user ratings of android apps. IEEE Trans Softw Eng 41(4):384–407

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 2016. ACM, pp 109–120

Chen X, Abdalkareem R, Mujahid S, Shihab E, Xia X (2019) Helping or not helping? Why and how trivial packages impact the npm ecosystem. Zenodo. https://doi.org/10.5281/zenodo.3417393

Cox R (2019) Surviving software dependencies. Commun ACM 62(9):36–43

DeBill E (2019) Modulecounts. http://www.modulecounts.com/#. Accessed 16 Jan 2019

Decan A, Mens T, Grosjean P (2018) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir Softw Eng

Fard AM, Mesbah A (2017) Javascript: the (un)covered parts. In: 2017 IEEE international conference on software testing, verification and validation (ICST), pp 230–240

Fleiss JL, Levin B, Paik MC (2013) Statistical methods for rates and proportions. Wiley, New York

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 24 Feb 2017

Gharehyazie M, Ray B, Filkov V (2017) Some from here, some from there: cross-project code reuse in github. In: Proceedings of the 14th international conference on mining software repositories, MSR '17. IEEE Press, pp 291–301

Jansen S, Brinkkemper S, Cusumano MA, Jansen S, Brinkkemper S, Cusumano MA (2013) Software ecosystems: analyzing and managing business networks in the software industry. Edward Elgar Publishing, Incorporated

Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empir Softw Eng 23(1):384–417

Linares-Vásquez M, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D (2014) How do api changes trigger stack overflow discussions? a study on the android sdk. In: Proceedings of the 22nd international conference on program comprehension, ICPC 2014. ACM, pp 83–94

Linares-Vásquez M, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D (2014) How do api changes trigger stack overflow discussions? A study on the android sdk. In: Proceedings of the 22nd international conference on program comprehension. ACM, pp 83–94

Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) Déjàvu: a map of code duplicates on github. Proc ACM Program Lang 1(OOPSLA)

MacDonald F (2018) How a programmer nearly broke the internet by deleting just 11 lines of code. https://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code. Accessed 09 June 2020

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. Ann Math Stat 18(1):50–60. (11 pages)

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 2017. IEEE Press, pp 84–94

npm Documentation (2020) npm-registry — npm documentation. https://docs.npmjs.com/using-npm/registry.html. Accessed 10 June 2020

Orsila H, Geldenhuys J, Ruokonen A, Hammouda E-B, Imed, Damiani E, Hissam S, Lundell B, Succi G (2008) Update propagation practices in highly reusable open source components. In: Open source development, communities and quality. Springer, US, pp 159–170

Sawant AA, Robbes R, Bacchelli A (2018) On the reaction to deprecation of clients of 4 + 1 popular java apis and the jdk. Empir Softw Eng 23(4):2158–2197

Scholtz A, Mehrotra P, Naumenko G (2018) Detection and mitigation of security vulnerabilities, pp 1–9

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

Serebrenik A, Mens T (2015) Challenges in software ecosystems research. In: Proceedings of the 2015 European conference on software architecture workshops, ECSAW '15. ACM, pp 40:1–40:6

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

StackOverflow (2020) Stack overflow developer survey 2020. https://insights.stackoverflow.com/survey/2020/. Accessed 09 June 2020

Tool SU (2020) Scitools.com. https://scitools.com/. Accessed 10 June 2020

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 40th international conference on software engineering, ICSE 2018. ACM, pp 511–522

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: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2018. ACM, pp 644–655

Vasilescu B, Blincoe K, Xuan Q, Casalnuovo C, Damian D, Devanbu P, Filkov V (2016) The sky is not the limit: multitasking across github projects. In: 2016 IEEE/ACM 38Th international conference on software engineering, ICSE 2016. IEEE, pp 994–1005

Wikipedia (2018) Unix philosophy - wikipedia. https://en.wikipedia.org/wiki/Unix_philosophy. Accessed 11 Jan 2019

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 2016. ACM, pp 351–361

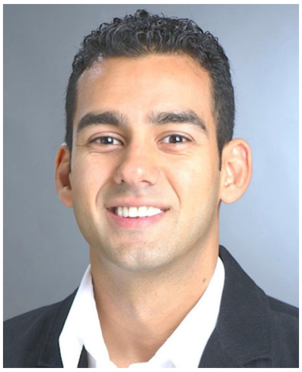
Yin RK (2009) Case study research: design and methods (applied social research methods). Sage, London and Singapore

Zimmermann M, Staicu C-A, Tenny C, Pradel M (2019) Small world with high risks: a study of security threats in the npm ecosystem. In: Proceedings of the 28th USENIX security symposium (USENIX Security, USENIX 2019. USENIX Association

Xiaowei Chen is a master's student at the Department of Computer Science and Software Engineering at Concordia University under Dr. Emad Shihabs supervision. Her research interest focus on software maintenance and evolution.

Rabe Abdalkareem is a postdoctoral fellow in the Software Analysis and Intelligence Lab (SAIL) at Queens 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, MSR, ICSME and MobileSoft, as well as in major journals such as TSE, IEEE Software, EMSE and IST. Contact him at abdrabe@gmail.com; https://rabeabdalkareem.github.io/.

Suhaib Mujahid 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, where his work focused on the detection and mitigation of permission-related issues facing wearable app developers. He did his Bachelors in Information Systems at Palestine Polytechnic University. His research interests include software ecosystems, software quality assurance, mining software repositories, and empirical software engineering. You can find more about him athttp://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

Xin Xia is an ARC DECRA Fellow and a lecturer at the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. Xin received both of his Ph.D and bachelor degrees in computer science and software engineering from Zhejiang University in 2014 and 2009, respectively. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information.More information at: https://xin-xia.github.io/.