On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages

Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, Senior Member, IEEE and Bram Adams, Senior Member, IEEE

Abstract—Nowadays, developing software would be unthinkable without the use of third-party packages. Although such code reuse helps to achieve rapid continuous delivery of software to end-users, blindly reusing code has its pitfalls. For example, prior work has investigated the rationale for using packages that implement simple functionalities, known as trivial packages. This prior work showed that although these trivial packages were simple, they were popular and prevalent in the npm ecosystem. This popularity and prevalence of trivial packages peaked our interest in questioning the ‘triviality of trivial packages’. To better understand and examine the triviality of trivial packages, we mine a large set of JavaScript projects that use trivial npm packages and evaluate their relative centrality. Specifically, we evaluate the triviality from two complementary points of view: based on project usage and ecosystem usage of these trivial packages. Our result shows that trivial packages are being used in central JavaScript files of a software project. Additionally, by analyzing all external package API calls in these JavaScript files, we found that a high percentage of these API calls are attributed to trivial packages. Therefore, these packages play a significant role in JavaScript files. Furthermore, in the package dependency network, we observed that 16.8% packages are trivial and in some cases removing a trivial package can impact approximately 29% of the ecosystem. Overall, our finding indicates that although smaller in size and complexity, trivial packages are highly depended on packages by JavaScript projects. Additionally, our study shows that although they might be called trivial, nothing about trivial packages is trivial.

Index Terms—Trivial Packages, npm ecosystem, Mining Software Repository.

1 INTRODUCTION

The use of third-party packages is becoming increasingly popular since it allows teams to reduce development time and costs and increase productivity [1], [2], [3]. A major enabler for the use of third-party packages (hereafter referred to as packages) is the capability for developers to easily share their code through software packages on dedicated platforms, known as software package managers (e.g., Node Package Manager (npm) and Python Package Index (PyPI)). Entire ecosystems have been created around these package managers, e.g., the Node.js ecosystem is largely supported by npm [4].

Despite the many benefits and widespread popularity of using software packages, they also pose some major drawbacks such as increased maintenance costs, an increased risk of exposure to vulnerabilities and even legal issues [5], [6], [7], [8]. One specific incident, the left-pad incident [3], [9], triggered a large debate on whether developers should be reusing packages for “trivial tasks”. Since then a number of studies focused on the topic of “trivial packages” and found that indeed, the left-pad incident is not isolated, and that trivial packages account for more than 17% of the 800,000 packages on npm [3], [10]. In addition, these packages tend to be heavily used, with some trivial packages (e.g., escape-string-regexp) being downloaded more than eleven million times per week [11].

The fact that these trivial packages play such a central role made us ask the question are trivial packages really trivial? Although we do agree that these packages may be small in size and implement very specific functionality, the fact that they are so prevalent is something that warrants the questioning of their triviality. Therefore, in this paper, we examine triviality of trivial packages based on their usage. In particular, we focus on the usage of trivial packages in 1) the projects that use them (project usage) and 2) the role they play in the ecosystem they belong to (ecosystem usage).

We perform an empirical study by analyzing more than 15,000 JavaScript projects, of which 3,965 depend on trivial packages. To examine project usage, we use static analysis to determine the centrality of the files that use trivial packages and analyze how widely the trivial packages are used in these files. To examine ecosystem usage, we leverage network analysis to examine the role of trivial packages in the ecosystems dependency network. Our study is formalized through three Research Questions (RQs):

- Project usage, RQ1: Are trivial packages used in central parts of JavaScript projects? To better understand how projects use trivial packages, we examine their role in the source code files of the projects that depend on them. Using the [Rabe: call graph], we find that files that depend on trivial packages are very central in their respective projects. This finding indicates that trivial packages may...
not be so trivial after all, since they are used in central parts of the projects that depend on them.

- **Project usage. RQ2: How widely used are trivial packages in JavaScript projects?** In addition to knowing if the trivial packages are used in central parts of the projects, we would like to know if the trivial packages are widely used (i.e., are they only used in one central part or throughout the projects). Again, we use static source code analysis to determine the percentage of API calls that are made to trivial packages. Also, we measure the entropy of the package to determine how widespread its use is. We find that trivial packages are at least as widely used as non-trivial packages, indicating that they may not be so trivial.

- **Ecosystem usage. RQ3: Do trivial packages play a central role at the ecosystem level?** To complement our analysis in RQs 1 and 2, which focus on project-level usage, we examine the centrality of trivial packages within the ecosystem. We study the package [Rabe: dependency network] for both direct and transitive dependencies of the studied projects. We find that trivial packages are more central to the ecosystem than non-trivial packages. Furthermore, we find that removing certain trivial packages from the ecosystem may impact up to 29% of the packages in the ecosystem. Our result shows that npm trivial packages are central building blocks in the ecosystem, and hence, their role is not so trivial.

Our study makes the following contributions:

- To the best of our knowledge this is the first in-depth study that examine the centrality and role of trivial packages to projects using them and to the ecosystem they belong to.
- The findings of this paper are based on an extensive analysis, which includes a large dataset of JavaScript projects that depended on trivial packages and the use of state-of-the-art technique that include [Rabe: dependency network analysis].
- To encourage replication and further study on use of trivial package, we have disclosed our dataset and source code for our analysis in our replication package.[Rabe: I have to add the dataset here!]

**Paper organization:** Section 2 presents our study design and approach. We describe our results in section 3. We discuss the results and implications of our study in Section 4 and Related works in Section 5. Threats to validity is shown in Section 6. Finally, Section 7 concludes our paper.

## 2 Case Study Design

To investigate the role of trivial packages in software projects, we study a large dataset of JavaScript software projects that depend on at least one npm trivial package.

![Diagram](chart.png)

**Figure 1:** An overview of our data collection approach.

---

| Filtering Step | # Projects |
|----------------|------------|
| JavaScript projects in GitHub | 7,863,361 |
| npm projects in GitHub | 2,289,130 |
| JavaScript projects that are not npm packages | 1,960,727 |
| Filtering out immature and/or inactive projects | 15,254 |

**Table 1:** Filtering steps of the studied JavaScript projects.

---

| Measurement   | Min. | Median | Mean   | Max.  |
|---------------|------|--------|--------|-------|
| Developers    | 2    | 5      | 6.74   | 69    |
| Commits       | 100  | 271    | 669    | 97,504|
| Watchers      | 1    | 6      | 23.99  | 2,451 |
| Stars         | 1    | 9      | 303.73 | 48,765|

**Table 2:** Summary of the number of developers, commits, watchers, and stars for 15,254 JavaScript projects.

---

**2.1 Dataset of Candidate Projects**

Since our analysis focuses on understanding the role of trivial packages in software projects that use them, we need to study a diverse and sufficiently large number of JavaScript projects that depend on trivial packages.

To acquire our dataset, we resort to the public GitHub torrent dataset [12], [13] to extract information about all the JavaScript projects hosted on GitHub. We extract the data pertaining to 7,863,361 JavaScript projects that are hosted on GitHub, as of 15th March 2019. We then filter out projects that do not use npm as their package management system. As a result, we found 2,289,130 projects that use npm as their package management system (i.e., projects have package.json file, which is the configuration file for npm projects). Moreover, since some npm packages use GitHub as their code repository [14], we exclude these npm packages from our list by crosschecking our list of URLs and GitHub URLs of all the npm packages. It is important to note that we exclude npm package repositories from our dataset so we do not analyze them as standalone JavaScript projects. We identify 328,343 npm packages in our list of candidate projects and we filter these packages out.

**2.2 Pruning List of Projects**

As recommended in prior work [3], [15], we perform extra steps to eliminate immature projects from our candidate dataset. To do so, we adopt similar filtering criteria that were used in prior work [3], [15]. We choose to select projects that are non-forks, have more than 100 commits by more than one contributor and have a community interest in them (i.e., projects that have at least one star and a watcher on GitHub). Finally, we select the projects that have at least
one external npm package dependency. These filtering steps allow us to extract a list of 15,254 JavaScript projects that are the client of npm packages (step 2 Figure 1). Table 1, shows the steps and number of projects after each step in the dataset acquisition process. Table 2 shows the summary statistics for different metrics of the selected JavaScript projects in our candidate dataset. As the table shows, our dataset contains a good distribution of projects in terms of developers, commits, watchers and stars.

2.3 Identifying JavaScript Projects that Use Trivial Packages

Since the goal of this study is to understand the role of trivial packages in JavaScript projects, we need to identify projects that depend on trivial npm packages in the selected candidate projects. To do so, we start by cloning the selected 15,254 projects. Then, we analyze them following a four-step approach (step 3 in Figure 1) to identify projects that use trivial packages.

First, we extract each project’s dependency information by examining the package.json file, which is the configuration file for npm projects. The package.json, among other configurations, specifies the list of packages that the project depends on. We extract the package name and its associated version for each runtime dependency for each project in our 15,254 projects candidate dataset. [Rabe: We only consider runtime dependencies since they are required to install and run the projects.]

Once we have the list of dependencies for each project in our candidate dataset, we download these packages using the package name and related version information. We download the dependent packages by using the npm-pack command [16]. The npm-pack command consults with the npm registry [17] and resolves the semantic version and downloads the appropriate ‘tar’ file that contains the source code of the package for each dependency-version pair.

Third, once we have the ‘tar’ file for each npm package, we analyze them to identify trivial packages. To do so, we extract the ‘tar’ file and analyze if the package is trivial or not by leveraging the definition proposed by Abdalkareem et al. [3], which categorize a package as trivial if its number of JavaScript “Line of code (LOC) ≤ Abdalkareem et al. [3] is trivial or not by leveraging the definition proposed by Abdalkareem et al. [3]. If a file depends on one or more trivial packages, we consider that file as a trivial dependent file, otherwise we consider it as a non-trivial dependent file. In the same way, if an project has at least one trivial dependent file then we identify it as a trivial dependent project.

According to this approach, in our candidate dataset, among the 15,254 JavaScript projects that we analyze, 26% (3,965) of the projects are trivial dependent. Since we want to analyze the role of trivial packages in JavaScript projects, we conduct our analysis on these 3,965 JavaScript projects dataset that use at least one trivial npm package. Table 4 shows the distribution of trivial and non-trivial packages in the projects in our dataset.

3 CASE STUDY RESULT

This section presents the results to our three RQs. For each RQ, we provide a motivation, describe the approach used and present our results.

3.1 RQ1: Are trivial packages used in central parts of JavaScript projects?

Motivation: Previous work showed that trivial npm packages are widespread, and has arguably some negative impact on software projects [3]. However, since these packages are small in size and complexity, one may expect that they are used in unimportant parts of software projects. To understand how projects use trivial packages, we examine their role in the source code files of the dependent projects. For example, if a trivial package is used in isolated part (i.e., file) in an project then its impact on that project can be neglected. Answering this question will help us understand the relative importance of trivial packages in the software projects that use them.

Approach: To examine a trivial package’s importance in a JavaScript project, we identify the files that use trivial packages since they provide a direct link between trivial packages and their importance in an project. [Rabe: I should remove this sentence. In this analysis, a trivial dependent file is a file that uses at least one trivial package, whereas, a non-trivial dependent file is a file that does not use any trivial packages.]

We examine the importance of trivial dependent files by analyzing the dependency call graph among the files of an project and measure the centrality score [20] of trivial dependent and non-trivial dependent files. To identify the JavaScript files that are more central in a software project, we apply network analysis on the call graph of each project and measure the centrality score. The centrality score of a node in a network reflects how important that node is in the network [21], [22], [23]. In scientific literature, network analysis is a popular measure in social sciences, which studies networks between humans (actors) and their interactions (ties). In our context, the JavaScript files are the actors and their inter-dependencies are the ties. For each JavaScript file within an project, we extract information on which other files the concerned file depends on (out-degree) and by which other files the concerned file is being dependent upon (in-degree). Then, we calculate the degree centrality score [20] for each file of an project in our dataset. The degree centrality score is a measure of the number of in-degree and out-degree for a JavaScript file within an project. This degree centrality score is normalized by dividing by $n - 1$, the maximum possible degree in a graph that has $n$ total nodes in that graph. The degree centrality of a node $V_i$ is given by:

\[
\text{Degree Centrality } (V_i) = \frac{|N(V_i)|}{n - 1}
\]
Table 3: Distribution of “Line of Code” and “Cyclomatic Complexity” of all packages (trivial and not-trivial packages).

| Type of packages | Line of Code |                |                |                | Cyclomatic Complexity |                |                |
|------------------|--------------|----------------|----------------|---------------------|-----------------------|----------------|----------------|
|                  | Min. | Median | Mean | Max. | Min. | Median | Mean | Max. |
| Trivial          | 7    | 19    | 19.35 | 34   | 1    | 5     | 5.324 | 10 |
| Non-trivial      | 36   | 200   | 2044.7 | 853967 | 2   | 50    | 699.58 | 313291 |

Where the $|N(V_i)|$ is the number of nodes (files in our case) that are connected to the node $V_i$ (i.e., file under examination). The degree centrality score has a value that ranges between 0-1, where 1 means that the node is in the center of the network (i.e., connected to all other nodes) and zero indicates that the node is isolated.

To calculate the degree centrality of trivial and non-trivial dependent files in each project in our dataset, we start by generating a call graph representation of files in every project. We use the madge tool to generate the call graphs [24]. The output of this tool is a call graph that shows each file in a software project and a list of files it depends on. We then run the networkx tool [25] on the generated call graph, to calculate the centrality score of every file in the graph. The networkx tool is a well-known tool for analyzing and visualizing social network data. Finally, to put our results in perspective, we compare and contrast the degree centrality score for trivial and non-trivial dependent files.

In addition, to get an in-depth understanding of the JavaScript file’s relative importance within a software project, we rank the files based on their degree centrality score, e.g., JavaScript file with highest degree centrality score is ranked 1 and the rank increases with decreasing degree centrality values. [Rabe: In case if two JavaScript files have similar degrees of centrality scores, they will have the same ranking.] Since the trivial dependent projects in our dataset vary in the number of JavaScript files, we segment the projects into four groups (based on the quartile they fall in) namely small, small-mid, mid-large, and large projects based on the distribution of the number of JavaScript files in the projects. From the distribution of the number of files in the studied projects, shown in Table 5, we group projects having $\#files < 1$st Qu. into small projects; $1$st Qu. $\leq \#files < \text{median}$ into small-mid projects; median $\leq \#files < 3$rd Qu. into mid-large projects, and $\#files \geq 3$rd Qu. into large projects. In addition, to put our results in perspective, we again compare the distribution of degree centrality rank for trivial dependent and non-trivial dependent files in each group of projects.

Results: Table 6 shows the summary distribution of the degree centrality score for trivial and non-trivial dependent files in our dataset. Here, we observe that overall the degree centrality values for trivial dependent files are higher than that of non-trivial dependent files. The table shows that the median/mean degree centrality values are 0.022/0.061 and 0.003/0.021 for trivial and non-trivial dependent files, respectively. To test if the difference is statistically significant between the two result sets, we applied the nonparametric Wilcoxon rank-sum test [26]. We determine if the difference is statistically significant at the customary level of 0.01. We also estimated the magnitude of the difference between datasets using the Cliff’s Delta [27] (or $d$). Cliff’s Delta is a non-parametric effect size measure for ordinal data. We consider the effect size values: negligible for $|d| < 0.147$, small for $0.147 \leq |d| < 0.330$, medium for $0.330 \leq |d| < 0.474$ and large for $|d| \geq 0.474$. We found that the results are statistically significant ($p$-value < 2.2e-16) with medium effect size ($d = 0.3471$).

In addition, Figure 2 shows a beanplot distribution of the degree centrality rank of trivial dependent and non-trivial dependent files in different project groups based on number of files.

Table 6: The distribution of the degree centrality of trivial and non-trivial dependent JavaScript files.

| File Type | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|-----------|------|---------|--------|------|---------|------|
| Trivial   | 0.00 | 0.003   | 0.022  | 0.061| 0.070   | 1    |
| Non-Trivial | 0.00 | 0.000   | 0.003  | 0.021| 0.019   | 1    |

Figure 2: Distribution of degree centrality rank of trivial dependent and non-trivial dependent files in different project groups based on number of files.

Table 4: Distribution of the number of npm packages that are used in all the JavaScript projects in our dataset.

| Type of packages | Min. | Median | Mean | Max. |
|------------------|------|--------|------|------|
| Trivial          | 1    | 2      | 2.34 | 31   |
| Non-trivial      | 1    | 16     | 19.69| 106  |

Table 5: Distribution of number of files in projects in our dataset.

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|------|---------|--------|------|---------|------|
| 10   | 26      | 54     | 115.1| 109     | 5921 |
Table 7: Distribution of number of trivial and non-trivial dependent files all the JavaScript projects in our dataset.

| File Type | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|-----------|------|---------|--------|------|---------|------|
| Trivial   | 1    | 1       | 1      | 3.313| 3       | 161  |
| Non-Trivial| 1    | 5       | 14     | 31.11| 33      | 1,592|

Figure 3: Distribution of percentage of API calls for trivial and non-trivial packages in JavaScript files.

Our findings indicate that trivial packages are used in more important and central parts of software projects compared to non-trivial packages. In our dataset, trivial dependent files have on median 0.022 degree centrality value while it is 0.001 for non-trivial dependent files. This difference is statistically significant.

3.2 RQ2: How widely used are trivial packages in JavaScript projects?

Motivation: Thus far, we saw that trivial packages are used in important parts of the projects that depend on them. Next, we want to examine the diffusion of a used package across the projects. In another word, we want to examine whether trivial packages are used only in important parts of the projects or their usage is dispersed across different parts of the projects. For example, prior work showed that if the Application Programming Interfaces (API) of a package $Pkg_A$ are invoked less than APIs’ of another package $Pkg_B$ in a software project then this is a clear indication that $Pkg_B$ is more important than $Pkg_A$ in that specific project [28]. Thus, low usage of trivial package APIs’ in a JavaScript file suggests that, even if these packages are used in more important files, these package’s importance within that file is low. Therefor, we investigate how heavily a trivial package’s APIs are used within a JavaScript file to determine these package’s importance within the trivial dependent JavaScript files.

Approach: To determine how widely used trivial packages are within an project, we again perform a two-way complementary analysis. First, we measure the percentage of each package’s programing interface calls in a file that depends on an external package in our dataset. Then, we examine how widespread the use of a package is in each project. Specifically, we use static code analysis and calculate the following two measures:

Percentage of trivial package API calls in a trivial dependent file: Although, based on our definition, a trivial dependent file has at least one trivial package dependency, in fact, it can have any number of non-trivial package dependencies. In our dataset, the median number of trivial and non-trivial packages in trivial dependent files are 1 and 3, respectively. Therefore, since these files have a lower number of trivial package dependencies, we want to understand what percentage of total API calls in a trivial dependent file are associated with trivial packages. To do so, we use a static source code analysis tool to extract and measure all the occurrences of external package API calls in JavaScript files.

[Rabe: Specifically, we use the Understand tool [18], which is a source code analysis tool that provides various code metrics and has been extensively used in other work [29], [30]] Then, we calculate the percentage of a package’s API calls within a JavaScript file by accounting all the API calls in that file.

External package entropy: We again use the extracted information about the API calls of external packages to compute the entropy of the packages. In our study, the entropy of a package shows how widely the package is used in an project. The higher the entropy of a package (i.e., API usage spread across files), the more difficult it gets to uproot the package from the project. Similar to prior work [31], [32], we define the entropy of an external package as the distribution of API calls of that package across files. For example, in a JavaScript project, the package $Pkg_x$’s APIs are called 10 times in file $F_1$, 15 times in file $F_2$, and twice in file $F_3$, we calculate the entropy of the package $Pkg_x$ as $(-rac{10}{27} \log_2 10 - rac{15}{27} \log_2 15 - rac{2}{27} \log_2 2)$, which equal to 1.28. It is important to note that the higher the entropy value the more widespread is the usage of the package is in a JavaScript project and if a package is used only in a single file then its entropy is zero.

Result: Figure 3 shows the distribution of percentage of API calls for trivial packages and non-trivial packages within the trivial dependent files. Here, we observe that median value of percentage of API calls for trivial packages within trivial dependent files is higher than that of non-trivial packages with median of 11.76% and 7.69% calls, respectively. We also
examine whether the result is statistically significant and we also calculate the effect size. We found that the results are statistically significant ($p$-value $< 2.2e-16$) and effect size is small (Cliff’s delta estimate = 0.25). This API call analysis of trivial dependent files shows that trivial packages play important role in these files.

In the second part of analyzing this research question, we investigate the distribution of API calls of a trivial package across the project by computing its entropy. Figure 4 shows a bean-plot distribution of entropy scores for trivial and non-trivial packages. We observe that trivial and non-trivial packages have similar entropy score distribution with median entropy score equal to zero for both types of packages. Most of the packages (68.067%) in our dataset have zero entropy scores, which suggests that these packages are used in only a single JavaScript file in the studied JavaScript projects. This result is statically significant with $p$-value $< 1.789e-05$ but the effect size is negligible (Cliff’s delta estimate: -0.1119). [Rabe: The entropy score distribution of trivial and non-trivial packages indicates that trivial and non-trivial packages tend to be used in different ways, but these two types of packages are essential in software projects.]

3.3 RQ3: Do trivial packages play a central role at the ecosystem level?

Motivation: In previous research questions, we found that trivial packages are important components for the JavaScript projects that directly depend on them. However, npm packages, trivial or non-trivial, do not exist on isolation, they interconnect with other packages and they form what is known as the npm ecosystem. We believe that examining how important trivial packages are with the software ecosystem that they belong to will provide us with a general understanding of their importance. Thus, in this question, we seek to understand the importance of a trivial package in the dependency network of npm ecosystem, which consists of all direct and indirect dependencies of the studied projects.

Approach: To examine the importance of trivial packages from the npm ecosystem perspective, we extract all the dependencies (direct and indirect) for each JavaScript project in our dataset and construct its dependency network graph. To extract this package dependency graph, initially, we install and clone the projects’ dependencies by using the `npm install` command, which install the package version specified in package.json file. By doing so, all the direct and indirect dependencies of every project in our dataset are saved locally in the project’s home directory in a folder named “node_modules”. Then, we use the `npm ls` [33] command to list installed package and their inter-dependencies in json format. Subsequently, we merge all the dependency network graphs of all the projects in our dataset and compile a composite dependency network at a given point in time. Figure 5 depicts an example of the process of merging the dependency network graphs of two JavaScript projects (`Project_A` and `Project_B`). In our illustrating example, `Project_A` is directly dependent on `pkg_X` which in turn depends on `pkg_Y` whereas `pkg_Y` depends on `pkg_Z`. `Project_B` has two direct dependencies and one transitive dependency. Here, in the composite dependency network, dependency hierarchy is preserved while accommodating all the dependencies of both projects. We recursively apply this merging process on all the dependency network of all the projects in our dataset. As a result of this merging process, we get a composite package dependency network that consists of 32,319 connected packages. Then we analyze the source code of each package in the constructed dependency network and identify trivial and non-trivial packages. [Rabe: We find that 16.8% of 32,319 packages in the constructed dependency network are trivial packages.] After that, we use the composite packages dependency network to examine the importance of trivial packages in two complementary measures. First, we measure the importance of trivial packages within this dependency network using the PageRank algorithm [34]. Second, we study the importance of the trivial packages by measuring the Technical Bus Factor (TBF) of these packages. Similar to the idea of social bus factor, which measures the effect of removal of a developer from a project, the TBF measures the effect of the removal of a package from a dependency network [35]. In the following subsection, we describe how we measure these values for every package in our constructed graph.

PageRank of External Packages: PageRank score [34] of a node (packages in our case) indicates the importance of the node in a network. The more dependent on a node in a network the higher is its PageRank score. PageRank has a value range between 0-1. We calculated the PageRank score of every package (trivial and non-trivial) in our composite package dependency network. To do so, we use the well-known network analysis tool called `networkx` tool [25]. Then, to put our results in perspective, we compare the PageRank score of trivial and non-trivial packages.

Technical Bus Factor (TBF): To understand the effect of removing one trivial package from the package dependency network, we calculate TBF, which simulates the removal of a package from our constructed composite network. We then evaluate how many other packages, directly or indirectly dependent on the removed package, are affected. We calculate what percentage of 32,319 packages, which is the total number of packages in our dependency network, are affected by the removal of one package from the package dependency network. The higher a package’s Technical Bus Factor (TBF) value; the more vital that node is in the package dependency network.

Result: Figure 6 shows PageRank score distribution for trivial and non-trivial packages. We notice that the median PageRank score of trivial packages (1.71e-05) is higher than that of non-trivial packages (1.61e-05). This result is significant ($p$-value $< 2.2e-16$) and effect size is small (Cliff’s delta estimate: 0.1578). This result shows that many packages
functions, enhancement of JavaScript standard functionalities and cross-platform compatibility features.

First, the examined trivial packages provide some popular utility functions like checking objects e.g. has-flag, has-color, is-object, number-is-nan; string operations e.g. ansi-regex, strip-ansi; and object manipulation e.g. xtend, forEach. The second group of the examined trivial packages is used to enhance existing native functionality of the JavaScript engine. For example, process-nextick-args [36] extends the capability of process-nextick by enabling this function to accept arguments. Finally, we found some trivial packages provide functionalities that help developers to deal with the cross-platform compatibility. Since JavaScript code can be run on different types and versions of web browsers, these packages provide backward and forward compatibility. For example, isarray package [37] is a well-known package and in the dependency network it is ranked 2nd based on its TBF. It provides same functionality like the native Array.isArray. Array.isArray supports browsers with newer version, e.g. IE9+, Chrome 5+, Firefox 4+, Opera 10.5+ and Safari 5+. However, as this function is not supported in older versions of browsers, isarray package is widely used because it supports older browser versions that are not compatible with ECMAScript 5 or later. These types of packages that provide cross-platform compatibility are known as ponyfills and polyfills [38]. Whereas polyfills are prone to unexpected bugs as these pollute the global scope, ponyfills is the smarter alternative which exports functionalities as a module without exploiting global scope. 25% of top 20 trivial packages e.g isarray, debuglog, object-assign, pinkie-promise, number-is-nan, are ponyfills. Furthermore, 37.97% of all ponyfill solutions in npm are trivial packages [38], [39]. From this analysis we see that trivial packages are often the byproduct of compatibility ensuring efforts.

Additionally, this analysis of the top 20 trivial packages revealed that some developers have a proclivity of publishing trivial packages. For example, Sindre Sorhus [40], a famous open-sourcerer, who created Yeoman [41] and Awesome Project [42], collaborated 7 of the top 20 trivial packages. We examined all of his 1,148 packages in npm and surprisingly 55.14% of his published packages are trivial packages.

Table 8: The statistical summary of the distribution of technical bus factor (TBF) for the trivial and non-trivial packages in our composite dependency network.

| File Type | Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|-----------|------|---------|--------|------|---------|------|
| Trivial   | 0.00 | 0.0031  | 0.0155 | 3.5324 | 0.1918  | 36.8174 |
| Non-Trivial | 0.00 | 0.0031  | 0.0093 | 1.9480 | 0.0495  | 34.9485 |

Trivial packages are vital nodes in the package dependency network (i.e., ecosystem). In fact, our results show that 16.19% of trivial packages and only 9.27% of non-trivial packages have a TBF value greater than 15%.
### Table 9: The top-20 most impactful trivial packages measured by Technical Bus Factor (TBF).

| Packages                  | TBF  | Rank | Functionality                                                                 |
|---------------------------|------|------|-------------------------------------------------------------------------------|
| inherits                  | 36.82| 1    | Inherits one constructor’s prototype to another constructor.                  |
| isarray                   | 35.43| 2    | Checks if the object in the argument is an array.                             |
| process-nextick-args      | 34.15| 10   | Amends the functionality of process.nextTick, which defers a callback function until next eventloop, enabling it to accept arguments. |
| debuglog                  | 34.13| 11   | Shows debugging information in stderr.                                        |
| escape-string-regexp      | 32.26| 14   | Escapes special characters.                                                   |
| ansi-regex                | 32.00| 18   | Matches ANSI escape codes.                                                    |
| object-assign             | 31.90| 22   | Assigns values to objects.                                                    |
| strip-ansi                | 31.89| 24   | Removes ANSI escape codes from a string.                                     |
| indexof                   | 31.14| 49   | Returns index of an object in an array.                                       |
| foreach                   | 30.87| 59   | Iterates over the key value pairs of either an array or a dictionary like object. |
| pinkie-promise            | 30.54| 63   | Returns JavaScript promise object.                                            |
| is-object                 | 30.20| 64   | Checks if the argument is an object.                                          |
| get-stdin                 | 30.10| 65   | Get standard input as a string or buffer.                                    |
| xtend                     | 30.03| 68   | Extends an object by appending all of the properties from each object in a list. |
| has-flag                  | 29.77| 70   | Checks if function argument has a specific flag.                             |
| has-color                 | 29.67| 73   | Detects whether a terminal supports color.                                    |
| once                      | 29.65| 74   | Restricts a function to be called only once.                                  |
| graceful-readlink         | 29.02| 79   | Returns a file’s symbolic link.                                               |
| number-is-nan             | 28.91| 82   | Checks whether the value in the argument is undefined and its type is Number. |

#### 4 Discussion

In this section, we first discuss our findings concerning the importance of trivial package overtime in software projects. Then, we discussed the implications of our findings.

#### 4.1 Re-examining the Role of Trivial Packages Overtime

Our results were presented on a specific snapshot of the projects and their dependencies. Hence, in this subsection, we further investigate the validity of our findings over time.

In research questions 1 and 2, we focus on studying the importance of trivial packages from the usage perspective. To do so, we examine the current snapshot of the studied projects. Now, we want to examine the role of trivial package in the studied projects. We believe that examining usage of trivial package over time will provide us with a general overview of the usage of trivial packages compare to only examine the current snapshot. Also, an increment in the number of trivial dependent files overtime in a software project suggests these packages’ importance and developer’s reliance on these packages whereas decrement suggests otherwise.

First, we examine the evolution of the number of trivial dependent files over their development timespan of a project. Second, we analyze the evolution of percentage of trivial package API calls in trivial dependent files over the development timespan of software projects. To identify the development period in which an project has some trivial package dependency, we need to know the commit that introduced the first trivial package in an project. This commit is either the first commit in a software project or before this commit the project was non-trivial dependent. Since all the projects in our dataset use git as their source control system, we iterate each commit starting from the initial commit of a software project to check if the commit is adding any trivial package into a JavaScript file. When we encounter such commit, we break the iteration and mark and register that commit as a trivial introducing commit for that software project.

Trivial dependent projects in our dataset start being trivial dependent from the trivial introductory commit. We consider the development timespan of an project, which ranges from first trivial introductory commit till the latest commit as trivial dependent development timespan (TDDT). We segment this TDDT into 10 equal parts by the means of the total number of commits in this period. For each project, we count the total number of commits in its TDDT and take a snapshot at each 10th percentile commit. Therefore, this segmentation process provides 11 snapshot points for each project, which are at: first trivial introductory commit, 10% commit, 20% commit, 30% commit, 40% commit, 50% commit, 60% commit, 70% commit, 80% commit, 90% commit and latest commit. As module growth is a predicted phenomenon in the software development lifecycle [43], [44], [45], we measure the percentage of trivial dependent files to all files across an project’s TDDT not the raw number.

Figure 7 shows box-plots of the percentage of trivial dependent files in all the studied projects based on TDDT segmentations. Dotted horizontal line present overall median.

![Figure 7: The distribution of the percentage of trivial dependent files in all the studied projects based on TDDT segmentations.](image-url)
Table 10: The statistical summary of the distribution of external package API call percentage in JavaScript files throughout project's development lifespan. The table shows the distribution for trivial packages (TP) and non-trivial packages (NPT).

| Segments | Intro | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% |
|----------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|          | TP   | NTP | TP  | NTP | TP  | NTP | TP  | NTP | TP  | NTP |
| Min.     | 0.1  | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| Median   | 16.7 | 9.1 | 16.7| 9.1 | 16.7| 9.1 | 16.7| 8.6 | 16.7| 8.3 |
| Mean     | 30.1 | 16.4| 30.0| 16.4| 30.5| 16.3| 30.2| 16.3| 30.7| 15.9|
| Max.     | 100  | 96.6| 100 | 96.6| 100 | 96.8| 100 | 96.9| 100 | 97.3|

Here, we observe that the percentage of the number of trivial dependent files remain almost constant over time with approximately median percent of trivial dependent files equal to 20%. These results reflect the importance of and developer’s reliance on these trivial packages in software projects.

We further investigate the percentage of trivial packages' API calls in trivial dependent files throughout the concerned project’s TDDT. Table 10 the shows percentage of package’s API calls distribution in these files for each project across its TDDT. Once again, to put our analysis in perspective, for every TDDT segment, the table shows the percentage of trivial packages (TP) and non-trivial packages’ API calls.

From Table 10, we observe that the percentage of trivial package’s (TP) API calls is higher than that of non-trivial package’s (NTP) API call at each snapshot point in the projects development timespan. For example, at 30%’s TDDT, we see that trivial packages’ API calls is higher (with mean=30.5 and median = 16.7) that the percentage of API calls for the non-trivial packages (with mean = 16.3 and median = 9.1). We see similar results at the late of the development lifespan of the studied projects. As the table shows at 90%’s TDDT, we see that with 30.3/16.7 mean/median of API calls for trivial packages is higher than the ones for the non-trivial packages (15.8/8.3).

To examine whether the results are statistically significant, we perform the Wilcoxon rank-sum test and the Cliff’s Delta effect size test on the data from each segment. The last two rows of Table 10 shows p-value and the effect size between the percentage of trivial and non-trivial packages’ API calls for every TDDT. From Table 10, we see that these results are statistically significant and have small effect sizes in all the snapshot points. For example, at 30% TDDT, we found that the different between the percentage of the API calls for trivial and non-trivial packages are statically significant (p-value = <2.2e-16) and the effect size is small. This analysis shows that the percentage of API calls for trivial packages within trivial dependent files remains higher throughout the development timespan of the concerned software projects.

4.2 Implications of our findings

[Implications for the ecosystem: take care of these, they are not BS packages. Maybe consider a standard library. - for the devs of trivial: maintain these, they are critical (cite old work to say these are mostly not even tested). Also, when developing these packages they should not be treated as toy or temp (same as earlier).]

- For the app devs: dont overlook the importance of trivial packages. They are critical to ur app and other projects.]

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

5 Related Work

In this section, we discuss work related to our study, which is mainly related to software ecosystems. The software projects that belong to the same software ecosystem has been a research interest lately. Several studies examine software ecosystems to understand their characteristics and evolution (e.g., [46], [47], [48], [49], [50], [51]).

Recently, Abdalkareem et al. [3] studied an emerging code reuse practice in the form of small packages, trivial packages, in the npm ecosystem. Abdalkareem et al. [3] studied various aspects regarding trivial packages. They first defined the size and complexity of these packages and we adopt this definition in our study. Whereas their study was conducted upon understanding why developers use trivial packages, our study examines the importance of these trivial packages in software projects.

Several other studies examined direct and transitive dependencies of software projects. Wittern et al. [52] examined packages in npm ecosystem and they observed that 32.5% of the packages have 6 or more dependencies. Moreover, 27.5% of the packages in npm are core packages as they are largely dependent on. Fard et al. [53], in their study, evaluated changeability in npm projects and they showed that the average number of dependencies in these projects...
is 6 and the number is always in the growing trend. Kikas et al. analyzed the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems and showed that the number of transitive dependencies is 10 times higher than the number of direct dependencies and this scenario is growing exponentially [54]. Recently, Zimmermann et al. [55] systematically examine dependencies between packages, the maintainers responsible for packages in the npm while focusing on security issues. Their results show that individual packages could impact large parts of the npm ecosystem. They also reported that a very small number of developers are responsible for a large number of npm packages. In our study, we also see that direct dependencies are only the tip of the iceberg, whereas indirect dependencies make up the largest portion of a package dependency network. In our analyzed dataset of software projects, we found 10,507 distinct packages as direct dependencies to these projects whereas the package dependency network, which has direct and transitive dependencies of these software projects, has 32,319 packages.

Researchers have also investigated developers’ rationale behind selecting package for his/her software project. Surprisingly, Haenni et al. found that developers generally do not apply any logical reasoning while selecting the packages, they just use them to accomplish their task. Abdalkareem et al. [3] found that developers have biased perception about trivial packages, developers think that these packages are well tested. Moreover, after including third-party packages, developers are often too reluctant to update their dependencies, which improves functionalities and fixes security issues or bugs, of these packages. Kula et al. [56] observed that 81.5% of their studied projects have outdated dependencies, although these projects heavily depend on external packages. Their interviewing of developers reveals that they are often unaware of the security vulnerabilities of underlying dependencies and therefore they perceive updating dependencies not a necessity but additional work. The study of Wittern et al. [52] shows us that the package version number is not a good predictor of a package’s maturity. Therefore, to assist developers in updating dependencies, evaluating four software packaging ecosystems (Cargo, npm, Packagist and Rubygems), Degan et al. [57] proposed an evaluation based on the “wisdom of the crowds” principle to select appropriate semantic versioning constraints for their dependencies. These type of ecosystem-wide studies help to clarify various general misconceptions and mitigate bad practices in ecosystems.

Other studies examine the API usage of external packages. Mileva et al. [58] studied API usage patterns of external libraries to examine the popularity of external package APIs’. They used this popularity metric to determine if a package is successful or not. In addition, Holmes et al. [28] quantitatively analyzed how APIs are used. They consider the frequency of API use as popularity and importance of that API. Similar to these studies, we determine the importance of an external package by analyzing the percentage of its API calls in the files that depend upon those packages.

Overall, our study examines software projects that depend on at least one trivial package from the npm ecosystem. So, our study is focused on the characteristics of software projects that adhere to the same environment. This categorization helps us understand the ecosystem better and helps adhere to good practices and mitigate bad practices ecosystem-wide.

6 Threats to validity

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

6.1 Construct validity

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. In our study, we used several in-house and state-of-the-art tools and techniques. We used the Depchecker [19] tool to extract file-level dependencies, the madge tool [24] for generating call graph, and the Understand tool [18] for static analysis. Hence, we are limited by the accuracy of these tools. It is also important to note that our study consists of four million JavaScript files. Thus, it is ample time and resource consuming to manually check each file and these tools’ results. To mitigate the threats related to using these state-of-the-art tools, we randomly selected five projects from our dataset and manually cross-checked the output of these tools and in all cases, the tools produce the correct results. We also use the networkx [25] tool to generate the dependency graph of files of every JavaScript project. Again, our graph dependency network analysis may influence the accuracy of the generated graph. To alleviate these issues, we manually examine the generated call graphs for five projects in our dataset and found that these graphs represent the dependency structure between files in these projects.[Rabe: I have to update this after analysis by Atick]

To answer our second research question, we only captured the direct usage of external packages in our static code analysis. For example, a package “X” is imported (e.g. require statement) and assigned it to a variable “a” and later “a” is assigned to another variable “b”. We only tracked the external package usage with variable “a” and did not track “b”. We decide to examine the direct usage of these packages for two main reasons. First, this type of transitive assignment of a variable is very rare in JavaScript code as other work shows [59]. This is why we believe that this shortcoming does not significantly impact our findings. Second, if we miss some of the usages of external packages, we missed both trivial and non-trivial packages. As we contrast trivial and non-trivial package usage, this effect will not affect the result of the comparison.

[Rabe: In our analysis, we restore to use the npm-pack command to resolve the semantic version and download the appropriate ‘tar’ file that contains the source code of the package for each dependency-version pair at the time of our analysis. Thus, dependencies could be different depending on the time of the analysis.]
6.2 External validity

In this subsection, we discuss the generalizability of our findings. Our dataset only consists of JavaScript projects, which use npm as their package manager, hence our findings may not hold for projects written in other programming languages or use different package manager. However, npm mainly supports JavaScript projects and it is one of the largest and most rapidly growing software ecosystem [60]. In addition, our dataset is used in our study present only open source project hosted on GitHub that may do not reflect proprietary projects. Also, our initial dataset size is 15,254 JavaScript projects that use the npm package manager, which may not represent the whole population of JavaScript projects.

7 Conclusion

Code reuse in the form of small/trivial packages became prevalent in software development [3], [61]. We observe that these trivial packages, being small in size and complexity, provide various functionalities ranging from string manipulation to security. Thus it is important to understand whether these packages are trivially used or their usage in software projects transcends their triviality. In this paper, we empirically examine trivial packages relative importance their use cases from two point of views; from the projects usage and ecosystem usage. to do so, we analyze a large dataset of open-source JavaScript projects that depend on at least on trivial package.

We observe that trivial packages are used in important part of the examined software projects compare to non-trivial packages. Our results show that trivial dependent files have on median 0.022 degree centrality value while it is 0.001 for non-trivial dependent files. We also find that trivial packages have a higher percentage of total API calls of JavaScript files (11.76% and 7.69% for trivial and non-trivial packages have a higher percentage of total API calls files have on median 0.022 degree centrality value while it is 0.001 for non-trivial dependent files. We also find that trivial packages have a higher percentage of total API calls of JavaScript files (11.76% and 7.69% for trivial and non-trivial packages have a higher percentage of total API calls). As for the ecosystem usage, we examine the relative importance of trivial packages in the ecosystem they belong to where we analyze the dependency graph of the direct and transitive dependencies of software projects in our dataset. We observe that trivial packages are highly dependent upon packages in the npm ecosystem, which makes trivial packages salient in the ecosystem. In some case removing one trivial package from the npm ecosystem could effect up to 29% of the whole npm ecosystem.

We believe that there are several possible directions for future work based on our findings. First, we would like to develop an advanced technique to detect and evaluate the quality of trivial package in an ecosystem since our results reveal that trivial packages play a key role in the ecosystem. Second, we want to devise an automatic approach to identify trivial package so developers can be aware that the packages that they use are trivial. Finally, since our study examines only the importance of JavaScript packages, we would like to investigate the notion of triviality in other/more software ecosystems.

References

[1] S. Wagner and E. Murphy-Hill, Factors That Influence Productivity: A Checklist, Berkeley, CA, 2019, publisher='"Apress", pp. 69–84.
[2] E. Murphy-Hill, C. Jaspan, C. Sadowski, D. Shepherd, M. Phillips, C. Winter, A. Knight, E. Smith, and M. Jorde, “What predicts software developers’ productivity?” IEEE Transactions on Software Engineering, pp. 1–1, 2019.
[3] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, “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, ser. ESEC/FSE 2017. ACM, 2017, pp. 385–395.
[4] R. Cox, “Surviving software dependencies,” Commun. ACM, vol. 62, no. 9, pp. 36–43, Aug. 2019.
[5] W. C. Lim, “Effects of reuse on quality, productivity, and economics,” IEEE Software, vol. 11, pp. 23–30, 1994.
[6] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ibara, “Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages,” in Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 559–563.
[7] A. Decan, T. Mens, and E. Constantinou, “On the impact of security vulnerabilities in the npm package dependency network,” in 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), May 2018, pp. 181–191.
[8] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, “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, ser. ICSE ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 331–341. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337263.
[9] F. Macdonald, “A programmer almost broke the internet last week by deleting 11 lines of code - sciencealert,” http://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code, March 2016, (accessed on 06/03/2016).
[10] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, “On the impact of micro-packages: An empirical study of the npm javascript ecosystem,” 2017.
[11] npm search, “escape-string-regexp - npm,” https://www.npmjs.com/package/escape-string-regexp, (accessed on 10/02/2019).
[12] “The ghtorrent project,” http://ghtorrent.org/, (Accessed on 02/18/2019).
[13] G. Gousios, B. Vasilescu, A. Serberenik, and A. Zaidman, “Lean ghtorrent: Github data on demand,” in Proceedings of the 11th Working Conference on Mining Software Repositories, ser. MSR 2014. ACM, 2014, p. 384387.
[14] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, “On the diversity of software package popularity metrics: An empirical study of npm,” in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, ser. SANER 2019, 2019, pp. 589–593.
[15] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The perils of mining github,” in Proceedings of the 11th Working Conference on Mining Software Repositories, ser. MSR 2014. ACM, 2014, pp. 92–101.
[16] “npm-pack — npm documentation,” https://docs.npmjs.com/cli/pack.html, (Accessed on 02/17/2019).
[17] “npm-registry — npm documentation,” https://docs.npmjs.com/misc/registry, (Accessed on 02/17/2019).
[18] “Understand TNstatic code analysis tool,” https://scitools.com/, (Accessed on 02/17/2019).
[19] J. Li and D. Lukic, “depcheck - npm,” https://www.npmjs.com/package/depcheck, (Accessed on 02/17/2019).
[20] “Centrality - wikipedia.”
[21] S. White and P. Smyth, “Algorithms for estimating relative importance in networks,” in Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ser. KDD ’03. ACM, 2003, pp. 266–275.
[22] F. Cadini, E. Zio, and C.-A. Petrescu, “Using centrality measures to rank the importance of the components of a complex network infrastructure,” in Proceedings of the Critical Information Infrastructure Security. Springer Berlin Heidelberg, 2009, pp. 155–167.
[23] X. Qi, E. Fuller, Q. Wu, Y. Wu, and C.-Q. Zhang, “Laplacian centrality: A new centrality measure for weighted networks,” Information Sciences, vol. 194, pp. 240 – 253, 2012.
[24] “Madge- developer tool for generating a visual graph of your module dependencies,” https://www.npmjs.com/package/madge, (Accessed on 02/17/2019).
[25] “Networkx - network graph analysis,” https://networkx.github.io/, (Accessed on 02/17/2019).
Emad Shihab is an associate professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queens University. Dr. Shihab's research interests are in Software Quality Assurance, Mining Software Repositories, Technical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and ACM. More information can be found at http://das.encs.concordia.ca.

Bram Adams is an associate professor at Polytechnique Montreal, where he heads the Lab on Maintenance, Construction, and Intelligence of Software. His research interests include release engineering in general, as well as software integration, software build systems, and infrastructure as code. Adams obtained his PhD in computer science engineering from Ghent University. He is a steering committee member of the International Workshop on Release Engineering (RELENG) and program co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019.