A Preliminary Study on Self-Contained Libraries in the NPM Ecosystem

Pongchai Jaisri, Brittany Reid, Raula Gaikovina Kula

Abstract The widespread of libraries within modern software ecosystems creates complex networks of dependencies. These dependencies are fragile to breakage, outdated, or redundancy, potentially leading to cascading issues in dependent libraries. One mitigation strategy involves reducing dependencies; libraries with zero dependencies become to self-contained. This paper explores the characteristics of self-contained libraries within the NPM ecosystem. Analyzing a dataset of 2763 NPM libraries, we found that 39.49% are self-contained. Of these self-contained libraries, 40.42% previously had dependencies that were later removed. This analysis revealed a significant trend of dependency reduction within the NPM ecosystem. The most frequently removed dependency was babel-runtime. Our investigation indicates that the primary reasons for dependency removal are concerns about the performance and the size of the dependency. Our findings illuminate the nature of self-contained libraries and their origins, offering valuable insights to guide software development practices.

keywords

Libraries, Open Source, Software Engineering

Pongchai Jaisri
Information Science Graduated School, Nara Institute of Science and Technology, Nara, Japan,
e-mail: jaisri.pongchai.js3@is.naist.jp

Brittany Reid
Information Science Graduated School, Nara Institute of Science and Technology, Nara, Japan,
e-mail: brittany.reid@naist.ac.jp

Raula Gaikovina Kula
Information Science Graduated School, Nara Institute of Science and Technology, Nara, Japan,
e-mail: raula-k@is.naist.jp

1 Introduction

In software development, a software library refers to pre-built, reusable code modules or libraries that developers integrate into their projects to enhance functionality processes. The term dependency in the context of software libraries refers to the reliance of the main library on external modules to function correctly. These dependencies are crucial in ensuring the proper execution of the software by providing essential functionality. Nowadays, the widespread of libraries within modern software ecosystems creates complex networks of dependencies (e.g., NPM for JavaScript, PyPI for Python, and Maven for Java).

The practice of adding numerous dependencies by library maintainers can lead to dependency bloat [4]. This presents a challenge, as maintainers may lack clear visibility into which specific parts of the library utilize each dependency. Consequently, the risk associated with dependency usage may increase. The dependencies introduce several challenges. First, they exhibit fragility; complex inter-dependencies mean a single broken dependency can have cascading effects [7, 14]. Second, dependencies are outdated or abandonment [15]. Third, redundancy is prevalent. The lack of strict publishing guidelines within the ecosystem allows developers to create and distribute libraries with overlapping functionality [1–3]. Finally, dependencies can potentially lead to cascading issues in dependent libraries.

Software ecosystems present a vast landscape of potential research topics. Directly related to the focus of this paper are investigations into trivial packages, software reuse & dependency changed, and self-contained library.

One mitigation strategy is dependency reduction. Rather than addressing problems rooted in external dependencies, maintainers may opt to remove problematic libraries entirely. Maintainers have the flexibility to remove dependencies as needed. If the process of dependency removal continues until a library has zero dependencies, it becomes a self-contained library.

In this paper, we further classify self-contained libraries into two distinct categories base on the dependency history:

1. Always Self-Contained Libraries: The libraries that maintained self-contained status from their initial release to the latest version (e.g., get-stdin).
2. Become Self-Contained Libraries: The libraries that initially possessed dependencies but subsequently removed them (e.g., prettier).

Our idea in this paper is exploring the characteristic of self-contained library within NPM ecosystem, guideline with 3 research questions.

1. **RQ 1**: *To what extent is the reducing of dependencies a prevalent phenomenon?*

   Motivation: Answering this research question will quantify the prevalence of dependency reduction among libraries. This metric will provide insight into the popularity of self-contained libraries and their potential impact within the broader software ecosystem.

2. **RQ 2**: *Which types of libraries are most reduced?*

   Motivation: This research question aims to identify the specific types of dependencies that are most frequently removed. Understanding these trends will shed light on evolving practices in dependency management and illuminate potential implications for dependent libraries.

3. **RQ 3**: *What factors are frequently associated with libraries that have reduced?*

   Motivation: This question extends the findings of RQ 1 and RQ 2., establishes the prevalence of self-contained libraries, while RQ 2 reveals the dependencies most frequently removed. By synthesizing this data, we can begin to identify common characteristics among libraries that shed dependencies, ultimately leading to a better understanding of the factors influencing this phenomenon.

This study investigates the characteristics of self-contained libraries within the NPM ecosystem. We analyze a dataset of 2,763 NPM libraries to identify the prevalence of libraries that have undergone dependency reduction, transforming them into self-contained libraries. Notably, our findings reveal that 40.42% of self-contained libraries had dependencies that were subsequently removed. We further explore the most frequently removed dependencies (e.g., `babel-runtime`) and delve into the motivations behind dependency removal by examining associated Git commits. By analyzing these reasons, we identify the most prevalent factors driving dependency reduction. The most prevalent factors are concerns about the performance and the size of the dependency. This research enhances our understanding of the evolving relationship between dependency libraries and their dependents within the NPM ecosystem, shedding light on dependency management trends.

2 Related Work

2.1 Trivial packages

Within the NPM ecosystem, the use of small, often single-function, libraries has been observed [1, 7]. These so-called ‘trivial packages’ come with several disadvantages. Maintaining a large number of small, trivial packages can significantly increase a developer’s workload. Additionally, these packages can lead to complex dependency chains that are difficult to manage and resolve (often referred to as ”dependency hell”) [1, 2]. The abundance of trivial packages can make it challenging to find the right one, especially when multiple packages offer similar functionalities. This overabundance also contributes to redundant packages within the ecosystem. Finally, projects that incorporate many trivial packages tend to have longer installation and build times due to the increased number of dependencies.

2.2 *Software reuse & dependency changes*

Software reuse is the process of creating software system from existing software rather than building them from scratch [8]. Software reuse is a fundamental practice in software development, offering benefits such as improved quality and reduced effort. This practice involves a provider who creates reusable software components and users who integrate these components, or dependencies, into their own software.

Dependency changes are a common occurrence in software development and can sometimes introduce problems for dependent libraries. Breakage in dependent libraries can occur due to various reasons [14], including: feature modifications, incompatible provider versions, changes in object types, undefined objects, incorrect code semantics, failed provider updates, function renaming, or missing files.

Dependency change issues extend beyond the NPM ecosystem. For example, a study examining deprecation in Javadoc [9] involved the manual analysis of 374 deprecated methods across four major Java APIs to determine whether deprecation reasons were documented.

This work examines how self-contained libraries form. Previous research has focused on removing dependencies [5]. However, increased dependency removal creates greater potential for libraries to become self-contained.

3 Dataset Preparation

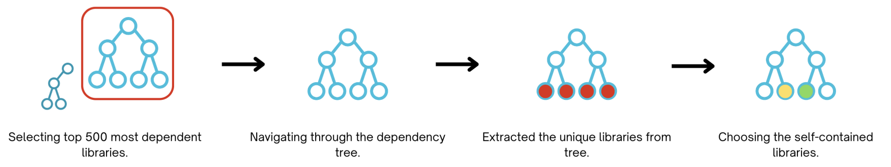
Selecting top 500 most dependent libraries.
Navigating through the dependency tree.
Extracted the unique libraries from tree.
Choosing the self-contained libraries.

Fig. 1 Overview of library selection

Figure 1 provides a visual overview of the library selection process for the dataset. We begin by selecting the top 500 libraries with the highest number of dependents from libraries.io[1], sorting them in descending order by dependent count. We then systematically analyze the dependency tree of each library by using the Open Source Repository and Dependency Metadata from libraries.io, yielding the dependency trees of 500 libraries. The dataset is available in Zenodo, at `https://doi.org/10.5281/zenodo.10972337` [6]

[1] `https://libraries.io/`

4 Empirical Study

In this section, we outline our approach to analyzing self-contained libraries, with a focus on understanding the motivations for removing dependencies. Our study is divided into three parts, each strategically designed to address a specific research question:

4.1 *RQ 1: To what extent is the reducing of dependencies a prevalent phenomenon?*

We investigated the dependency histories of self-contained libraries, utilizing the registry npmjs API[2] to collect historical dependency data which not later than year 2019. We focus exclusively on regular dependencies, excluding both devDependencies and peerDependencies. The devDependencies are necessary only during development and not included in production environments. The peerDependencies, while indicating potential compatibility with other libraries, may not have direct API interactions. To maintain clarity in this investigation, we limit our analysis to regular dependencies, avoiding the complexities introduced by other dependency types.

4.2 *RQ 2: Which types of libraries are most reduced?*

We conducted a detailed analysis of what dependencies were removed from the identified libraries. Utilizing the dependency history data provided by the registry.npmjs API, we compared dependency lists across versions. Our focus was specifically on versions where dependencies were reduced.

4.3 *RQ 3: What are the reasons for reducing libraries?*

Figure 2 provides a visual overview of gathering the Git commits of the libraries. For each dependent library, we examined its GitHub[3] repository to pinpoint the version where the maintainer reduced the target dependency. We collected relevant commits within that period. By matching pull requests with corresponding commits, we analyzed discussions within the pull requests and the associated code changes. Our goal was to identify the primary motivations driving the maintainer's decision to remove the target dependency.

[2] `https://registry.npmjs.org`
[3] `https://github.com/`

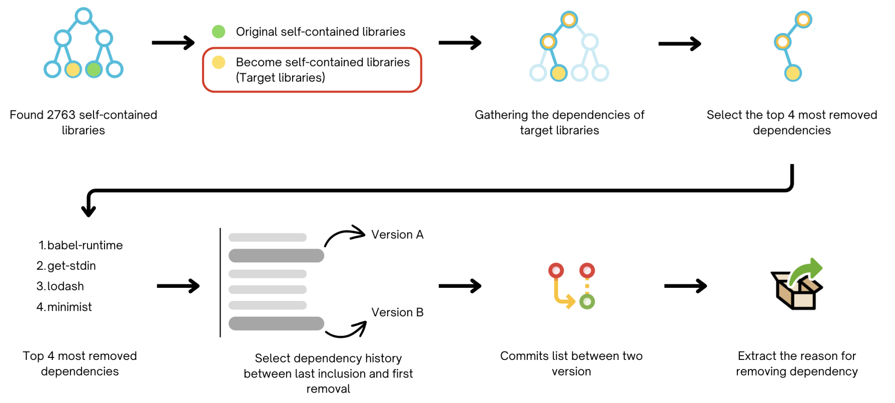
Found 2763 self-contained libraries
Original self-contained libraries
Become self-contained libraries (Target libraries)
Gathering the dependencies of target libraries
Select the top 4 most removed dependencies
1.babel-runtime
2.get-stdin
3.lodash
4.minimist
Top 4 most removed dependencies
Version A
Version B
Select dependency history between last inclusion and first removal
Commits list between two version
Extract the reason for removing dependency

Fig. 2 Overview of gathering target commits

Table 1 Statistic of the libraries and dependencies.

Library categories	**No. libraries (%)**	
Non Self-contained libraries	1,672	(60.51%)
Self-contained libraries	1,091	(39.49%)
- Become self-contained	441	(15.96%)
- Original self-contained	650	(23.53%)
All	2,763	(100%)

5 Result

5.1 *RQ 1: Prevalence*

Our analysis of the 500 most depended upon libraries revealed 2,763 unique libraries within the dependency chain. Further investigation into these libraries identified 1,091 as having no dependencies, or self-contained. Of the self-contained libraries, 441 achieved self-contained status through dependency removal. Table 1 provides a detailed statistical breakdown of these libraries and their dependencies.

Finding of RQ 1

We found 39.49% of unique libraries are self-contained libraries. Moreover, we found 15.96% of unique libraries became self-contained libraries; that is, they reduced dependencies to zero.

5.2 RQ 2: Most Reduced Libraries

From the 441 libraries that become self-contained libraries, we identified 407 dependencies that were removed. The most frequently removed dependencies were `babel-runtime`, `lodash._root`, and `lodash.keys`. Figure 3 reveals a significant trend in dependency reduction, highlighting the most commonly removed dependencies.

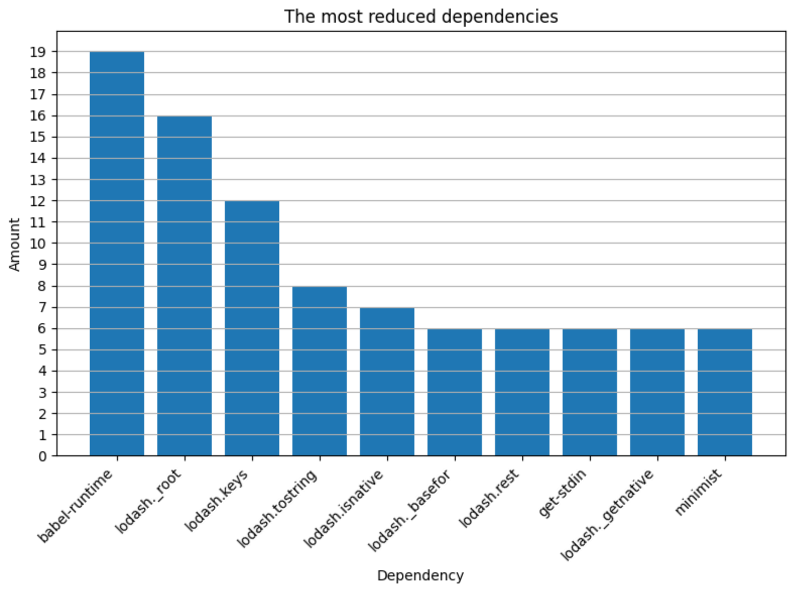

Fig. 3 The most reduced dependencies

Further analysis revealed that many of these reduced dependencies were, in fact, sub-dependencies (such as `lodash._root` and `lodash.keys`) from the bundled library `lodash`. We identified 87 sub-dependencies within the complete list of 407 reduced dependencies. After isolating these sub-dependencies, our revised analysis indicates that the top three most frequently removed dependencies are `babel-runtime`, `get-stdin`, and `minimist`. Figure 5 displays the most frequently removed package-type dependencies (excluding sub-dependencies) and the corresponding number of reductions and Figure 4 specifically details the most frequently removed `lodash` sub-dependencies.

Finding of RQ 2

The most reduced package dependency is `babel-runtime` while the most reduced sub-dependency is `lodash._root`.

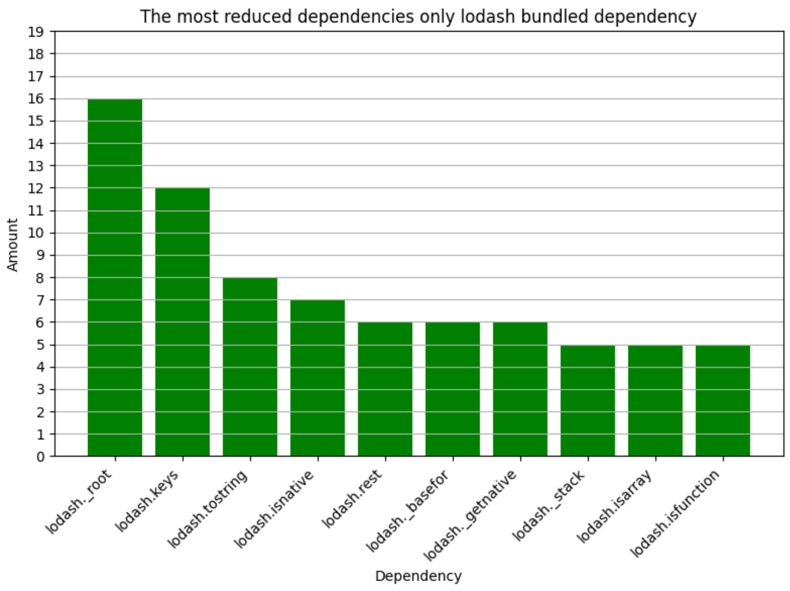

Fig. 4 The most reduced dependencies only lodash bundled dependency

Table 2 Number of the gathered commits.

Removed dependency	**No. commits**	**No. dependent libraries**
Single dependency	300	27
- babel-runtime	100	6
- get-stdin	100	15
- minimist	100	6
Bundled dependency	100	1
- lodash (7 sub-dependency)	100	1
All	400	28

5.3 RQ 3: Reasons for Reducing

Guided by the findings of RQ 2, we focus exclusively on reduced dependencies that are not sub-dependencies. Our analysis centers on the top three most frequently removed dependencies: `babel-runtime`, `lodash._root`, and `lodash.keys`, looking at 27 parent libraries. Consistent with our methodology, we distinguish between sub-dependencies from bundle dependencies and single dependenc. As in the Table 2, we examine 300 random commits from 27 dependent libraries where these three dependencies were reduced (normal dependencies), along with 100 random commits from 7 sub-dependency where bundled dependency were removed. Therefore, these are the top three most frequently removed dependencies after we split sub-dependencies from bundled library. We split 100 commits for each dependency.

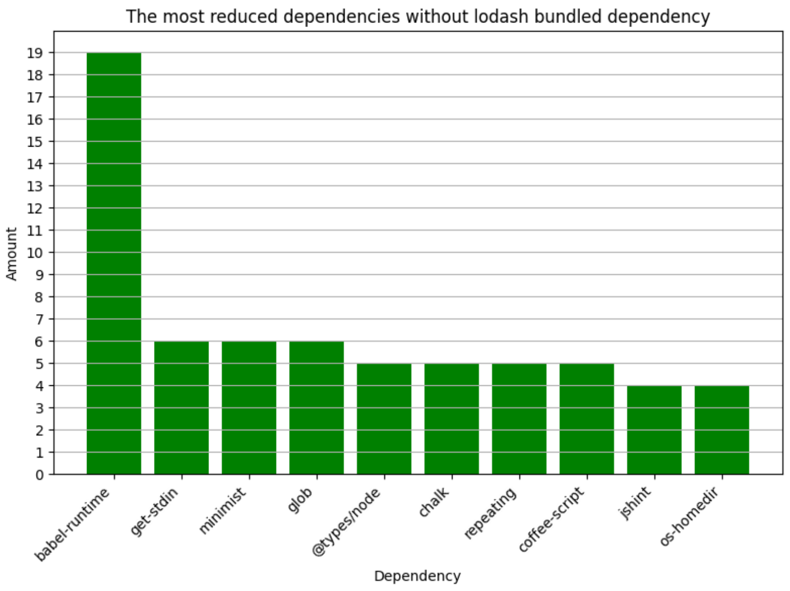

Fig. 5 The most reduced dependencies without lodash bundled dependency

Table 3 Categories of classifying reasons for 28 dependent libraries.

Terminology	**Definition**	**Libraries (%)**
Performance of the dependency	The dependency is too heavy, the maintainer of dependents library need to remove this dependency.	5 (17.86%)
Function replacement	Replace the dependency's function with built-ins function or custom function	3 (10.71%)
Dependency replacement	Replace the dependency with another dependency	1 (3.57%)
Minimize the dependency	The dependency is too heavy, the maintainer of dependency prefer to split some features of the dependency to independent library.	4 (14.29%)
Removing unused dependency	Remove unnecessary or unused dependency	1 (3.57%)
Others	The other reasons that cannot classify	14 (50%)

Below are the reason for the removal of dependencies that we found from random Git commit messages from dependent libraries.

Table 3 presents the terminology, definitions used to classify reasons for reducing software dependencies and the overall library with commit for each terminology. This classification criteria was developed through an analysis of commit messages, discussions within matched pull requests, and the relevant code modifications.

Results for the 28 libraries show that dependency removal was motivated by several factors: performance of the dependency, affecting 5 libraries (18.52%), function replacement affecting 3 libraries (11.11%), dependency replacement affecting 1 library (3.7%), minimalist the dependency affecting 4 libraries (14.81%), and removing unused dependencies affecting 1 library (3.7%), the others are 14 libraries (50%).

babel-runtime

We found the reasons for dependency-related changes from 3 distinct parent packages: `babylon`, `common-tags`, and `babel`.

1. babylon[4]: We found that the maintainer removed `babel-runtime` because they were using loose mode: a special mode from babel which helps to refactor the source code. *"remove babel-runtime dep/transform-runtime since we are using loose mode"*. We conclude that this reason is a replacement function.
2. common-tags[5]: we found a few reasons that the maintainer removed `babel-runtime` because this dependency is a heavy dependency. *"The issue in the article is with @angular/cli using 3 methods from common-tags, but the outlaying issue is babel-runtime is a heavy dependency."*. For this reason, we conclude that this reason is a performance of the dependency. The another reason is the maintainer replace the babel-runtime with Typescript which can handle the browser compatibility. *"We are considering dropping Babel and replacing it with Typescript, which would inline all the needed stuff. We care about browser compatibility, which is what Babel is used for, and Typescript is handling that well (even as far as ES3)."*. We conclude that this reason is a dependency replacement.
3. babel[6]: We found variety of reasons that the maintainer removed `babel-runtime`.

    - The first reason is they want to replace function from `babel-runtime` with built-ins function from the new version of Node. *"The reason for doing this in the first place has to do with wanting to use built-ins like Symbol/Promise, etc which are not native to node 0.10/0.12. Now that we are on ¿= Node 4 we should be able to use the native ones."*. We conclude that this reason is a function replacement.
    - The second reason is babel is a big dependency because deduping never works and the maintainer encountered with many copies of `babel-runtime`. *"Babel is always the biggest dependency in my projects because deduping never works and I end up with a gadzillion copies of babel-runtime."*. We conclude that this reason is a performance of the dependency.
    - The third reason is the maintainer thinking about less dependency and use native will make `babel` faster to run and uninstall.*"less dependencies, use native, might be faster to run/install"*. We conclude that this reason is a performance of the dependency.

[4] https://github.com/babel/babylon/pull/110
[5] https://github.com/zspecza/common-tags/pull/148
[6] https://github.com/babel/babel/issues/5118

get-stdin

We found the reasons for dependency removal from 5 difference dependent packages; `dateformat`, `pretty-bytes`, `indent-string`, `detect-indent`, and `detect-newline`.

1. dateformat[7]: We found that the maintainer removed `get-stdin` with other libraries. They concern about the flattened dependency tree of the dateformat. *"I love using this, but I just realized that it's using meow, and I'm guessing that a good percentage of users like myself probably aren't using the CLI, but the dependency tree for it is absurdly massive. Especially considering this would have zero deps without it. Meow has 48 dependencies in total, this is its flattened dependency tree."*. We conclude that this reason is a performance of the dependency.
2. pretty-bytes [10], indent-string [13], detect-indent [11], and detect-newline [12]: We found that these libraries have a same maintainer. The maintainer of these module split the function in the libraries into normal version and CLI version. The normal version used to have dependencies but the maintainer separate them into the CLI version. We conclude that this reason is a minimalist the dependency.

minimist

We found the reasons for dependency removal from 6 difference dependent packages; `flow-parser`, `envinfo`, `prettier`, `indent-string`, `detect-indent`, and `detect-newline`.

1. flow-parser[8]: The maintainer remove function name "flowparse" and "flowvalidate" from the library. Therefore, they removed the dependency which support those function. We conclude that the reason is a removing unused dependency.
2. envinfo[9]: The maintainer relocated minimist from regular dependencies to devDependencies, meant for development-only packages that are removed during the build process. We conclude that the reason is a performance of dependency.
3. prettier[10]: We found that the maintainer remove `minimist` because they want to transform `prettier` to dependency-free. *"Move all the dependencies to dev dependencies and –exact. Since we are now bundling all the dependencies, we can have prettier be dependency-free on npm ¡3"*. We conclude that this reason is a function replacement.
4. indent-string, detect-indent, and detect-newline: The reason is the same reason with removing `get-stdin` which is a minimalist the dependency.

lodash

Among removed dependencies classified as sub-dependencies from bundle dependencies, the top three most frequent were `lodash._root`, `lodash.keys`, and `lodash.tostring`. However, the maintainer did not provide explicit reasons for the removal of these specific sub-dependencies.

7 https://github.com/felixge/node-dateformat/issues/36
8 https://github.com/facebook/flow/pull/3586/files
9 https://github.com/tabrindle/envinfo/pull/29
10 https://github.com/prettier/prettier/pull/1850

Finding of RQ 3

The primary factor for removing dependency is *'the performance of the dependency'*, affecting more than 18% of the dependent libraries. This is followed by *minimize the dependency* (more than 14%) and the *function replacement* (more than 11%).

6 Discussion and Future work

In this work, we show that developers may be changing their attitudes on simply adopting blindly adopting dependencies into their applications. Based on the results of the study, one potential application is the development of a tool that recommends dependencies suitable for removal. Informed by the characteristics of frequently removed dependencies identified in this work, such a tool could analyze dependency lists (e.g., package.json in JavaScript, pyproject.toml in Python, etc.) and provide insights regarding individual dependencies. A tool of this nature could prove valuable to a broad range of users within the development community, extending beyond the immediate maintainers of self-contained libraries.

7 Acknowledgements

This work is supported by the Japanese Society for the Promotion of Science (JSPS) KAKENHI Grant Number JP20H05706.

References

1. Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad 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*, pages 385–395, 2017.
2. Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering*, 25:1168–1204, 2020.
3. Xiaowei Chen, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Xin Xia. Helping or not helping? why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering*, 26:1–24, 2021.
4. Ching-Chi Chuang, Luís Cruz, Robbert van Dalen, Vladimir Mikovski, and Arie van Deursen. Removing dependencies from large software projects: are you really sure? In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 105–115, 2022.
5. Ching-Chi Chuang, Luís Cruz, Robbert van Dalen, Vladimir Mikovski, and Arie van Deursen. Removing dependencies from large software projects: are you really sure? In *2022 IEEE 22nd*

International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 105–115, 2022.

6. Pongchai Jaisri. The commit history of the dependent libraries. https://doi.org/10.5281/zenodo.10972337, 2024.
7. Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *arXiv preprint arXiv:1709.04638*, 2017.
8. Johannes Sametinger. *Software engineering with reusable components*. Springer Science & Business Media, 1997.
9. Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. Why are features deprecated? an investigation into the motivation behind deprecation. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 13–24. IEEE, 2018.
10. Sindre Sorhus. extract CLI into a separate module. https://github.com/sindresorhus/pretty-bytes/commit/cdf9f2a6c14374dd4a648cf0b0517a714b97c820, 2024. [Online; accessed 9-March-2024].
11. Sindre Sorhus. extract CLI into a separate module. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2024. [Online; accessed 9-March-2024].
12. Sindre Sorhus. extract the CLI into a separate module. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2024. [Online; accessed 9-March-2024].
13. Sindre Sorhus. move CLI into a separate module. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2024. [Online; accessed 9-March-2024].
14. Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A Gerosa, and Igor Scaliante Wiese. I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–26, 2023.
15. Supatsara Wattanakriengkrai, Dong Wang, Raula Gaikovina Kula, Christoph Treude, Patanamon Thongtanunam, Takashi Ishio, and Kenichi Matsumoto. Giving back: Contributions congruent to library dependency changes in a software ecosystem. *IEEE Transactions on Software Engineering*, 49(4):2566–2579, 2022.