

Master's Thesis

An Empirical Study on Removing Library Dependencies for Open Source Libraries.

Pongchai Jaisri

September 4, 2024

Graduate School of Science and Technology
Nara Institute of Science and Technology

A Master's Thesis
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
MASTER of ENGINEERING

Pongchai Jaisri

Thesis Committee:

Professor Kenichi Matsumoto	(Supervisor)
Professor Keiichi Yasumoto	(Co-supervisor)
Visiting Professor Takashi Ishio	(Co-supervisor)
Associate Professor Raula Gaikovina Kula	(Co-supervisor)
Assistant Professor Shimari Kazumasa	(Co-supervisor)

An Empirical Study on Removing Library Dependencies for Open Source Libraries. *

Pongchai Jaisri

Abstract

The widespread use of libraries within modern software ecosystems creates complex dependency networks. Libraries are useful for software developers to develop software projects, allowing them to avoid starting from scratch. However, these dependencies can be fragile, outdated, or redundant, potentially leading to cascading issues in dependent libraries. One mitigation strategy is reducing dependencies when problems occur. This thesis investigates dependency removal in the NPM ecosystem. By analyzing a dataset of 52,115 commits from 2,763 NPM libraries, I found that 58.92% contained at least one dependency removal before December 31, 2019. During the analysis, I examined the dependency history of each library and filtered the commits related to dependency removal. I then matched these filtered commits with their corresponding pull requests. The reasons for dependency removal were analyzed through commit messages, code changes, and pull request titles and descriptions. The analysis revealed that the most frequently removed dependency was `object-assign` (59 times), with the primary reason being developers replacing the dependency with built-in functions or custom code. This finding highlights the origin of problems related to dependencies and offers valuable insights to guide software development practices. Additionally, this work suggests ideas for future research, such as creating tools to detect code that relies on specific dependencies within a project.

Keywords:

Library, Dependency, Dependent, Self-Contained library, Non-Reducing library, Transitional library

*Master's Thesis, Graduate School of Science and Technology,
Nara Institute of Science and Technology, September 4, 2024.

Contents

Abstract	i
List of Figures	iv
List of Tables	v
1 Introduction	1
1.1 Research Statement	2
1.2 Organization	5
2 Related Work and Background	6
2.1 Trivial Packages	6
2.2 Software Reuse	6
2.3 Dependency and Dependent Library	7
2.4 Dependency Changes and Updates	7
2.5 Dependency Network	8
2.6 Version Control System	8
3 Dataset Preparation	10
4 Empirical Study	12
4.1 RQ 1: To what extent is the reducing of dependencies a prevalent phenomenon?	12
4.2 RQ 2: Which types of libraries are most reduced?	12
4.3 RQ 3: What factors are frequently associated with libraries that have reduced?	13
5 Result & Discussion	15
5.1 RQ 1: Prevalence	15

5.2	RQ 2: Most Reduced Libraries	16
5.3	RQ 3: Reasons for Reducing	19
6	Implications	28
7	Threats to Validity	29
7.1	Internal Validity	29
7.2	External Validity	30
7.3	Construct Validity	30
7.4	Conclusion Validity	30
8	Conclusion	32
9	Future Work	34
	Bibliography	37
	Publication List	41

List of Figures

3.1	Overview of library selection	11
4.1	Overview of extracting the reasons for dependency removal	14
5.1	Top 10 most reduced dependencies	17
5.2	Types of dependency removal scenario	19
5.3	The Amount of Scenario Based on Retaining Features	24

List of Tables

5.1	Statistic of the libraries and dependencies.	15
5.2	Number of the gathered commits.	16
5.3	Top 3 most removed dependencies.	16
5.4	Terminology of classifying reasons from all dependency removal scenarios.	18
5.5	Top 5 dependencies replaced by built-ins or custom functions . . .	20
5.6	List of sub-dependency of lodash	25
5.7	The list of babel sub-dependencies	26
5.8	Example of Dependency with replace built-ins function or equivalent custom function	27

1 Introduction

In software development, a software ecosystem is a complex network of software applications, tools, libraries, frameworks, platforms, and communities that work together to create a unified and sustainable environment [11, 14]. For example, the NPM ^{*} for JavaScript, PyPI [†] for Python, and Maven [‡] for Java. Within these ecosystem, libraries play a crucial role by providing distributed code modules or components that developers can integrate into their projects, enhancing functionality and enabling them to avoid starting from scratch [18]. 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.

In modern software ecosystems, the widespread use of libraries creates complex networks of dependencies. The practice of adding numerous dependencies by library maintainers can lead to “dependency hell” [17]. This situation arises when maintainers lack clear visibility into which specific parts of the library utilize each dependency, increasing the risk associated with dependency usage.

Using dependencies introduce several challenges. First, installing too many dependencies can result in dependency bloat, making the software overly complex and difficult to manage [7, 24]. Second, dependencies exhibit fragility; complex inter-dependencies mean that a single broken dependency can have cascading effects throughout the software [10, 16, 25]. Third, dependencies can become outdated or abandoned, leaving the software vulnerable to issues [27]. Fourth, redundancy is a common problem due to the lack of strict publishing guidelines within the ecosystem, allowing developers to create and distribute libraries with overlapping functionality [1, 2, 5]. Additionally, dependencies may conflict with

^{*}<https://www.npmjs.com/>

[†]<https://pypi.org/>

[‡]<https://central.sonatype.com/>

each other [26]. Differences in the requirements of dependencies and the environment can lead to incompatibility issues during installation.

Furthermore, dependencies can be a root cause of vulnerabilities. As the number of dependencies increases, so does the risk of introducing vulnerabilities into the software [6, 19, 30]. Finally, dependencies can potentially lead to cascading issues in dependent libraries, exacerbating the impact of any single point of failure. These challenges underscore the importance of effective dependency management and the need for maintainers to be aware of the implications of their dependency choices.

1.1 Research Statement

Software ecosystems present a vast landscape of potential research topics. Directly related to the focus of this thesis are investigations into trivial packages, software reuse & dependency changed, dependency removal scenario, and the reason for dependency removal.

One mitigation strategy is dependency removal. 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. According to my previous study [13], I categorized libraries in the NPM ecosystem into two types based on their dependency usage characteristics:

1. Original Self-Contained Library: Libraries that have maintained a self-contained status from their initial release to the latest version (e.g., `get-stdin`).
2. Become Self-Contained Library: Libraries that initially had dependencies but subsequently removed them (e.g., `prettier`).

In this thesis, I have refined these categories to better reflect the dependency history of the libraries:

1. Original Self-Contained Library: Libraries that have maintained a self-contained status from their initial release to the latest version.

2. Non-Reducing Dependency Library: Libraries that have never decreased their dependencies and contain at least one dependency.
3. Transitional Library: Libraries that have a history of dependency removal. This category includes libraries that are neither original self-contained nor non-reducing dependency libraries. This type extends from the concept of become self-contained libraries by including those that have experienced at least one instance of dependency removal.

This thesis focuses on transitional libraries. The goal is to explore the characteristics of dependency removal in libraries within the NPM ecosystem, guided by three research questions.

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

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

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.

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

Motivation: This question extends the findings of first and second research questions. RQ 1 establishes the prevalence of libraries which contained dependency removal scenarios, while RQ 2 reveals the dependencies most frequently removed. By synthesizing this data, I 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 transitional libraries within the NPM ecosystem. By analyzing a dataset of 52,115 commits from 2,763 NPM libraries, I aim to identify the prevalence of libraries that involve dependency removal. These libraries are categorized into three types: become self-contained libraries, non-reducing dependency libraries, and transitional libraries

The findings reveal that 1,628 out of 2,763 libraries, or 58.92%, are transitional libraries. Furthermore, I explore the most frequently removed dependencies, such as `object-assign`, and examine the motivations behind these removals by analyzing associated Git commits and matched pull requests. This analysis helps identify the most common factors driving dependency removal, with the primary factor being developers replacing dependencies with built-in or custom functions. Interestingly, the study also highlights that maintainers often prefer to retain features even when dependencies are removed or replaced.

This thesis enhances our understanding of the evolving relationship between dependency libraries and their dependents within the NPM ecosystem. Maintainers of other dependencies can use the findings of this thesis as feedback to improve their libraries and provide better dependencies to others. Meanwhile, developers can leverage these insights to make informed decisions when selecting dependencies, thereby improving overall software quality.

In additions, this thesis also provide ideas for future research, such as developing a taxonomy of reasons for dependency removal, this idea will help other developers for the limit usages of each dependency based on the categories of reasons for dependency removal. The another idea is creating a tool to detect code that relies on specific dependencies within a project, this idea provides an automatic tool which can assist developers to decided which dependency can be removed with consequences effects to the project. By report which code rely on each dependency.

1.2 Organization

The organization of this thesis is as follows: Chapter 2 explores the background and related work related to this study. Chapter 3 explains the steps to prepare the dataset before analysis and how to replicate the dataset. Chapter 4 describes the step-by-step process for analyzing the dataset based on each research question. Chapter 5 presents the results of all research questions. Chapter 6 discusses the implications of this thesis. Chapter 7 examines the potential threats to the validity of the study. Chapter 8 summarizes the thesis, and Chapter 9 offers ideas for extending this research.

2 Related Work and Background

2.1 Trivial Packages

Within the NPM ecosystem, the use of small, often single-function, libraries has been observed [1, 16]. 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") [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

Software reuse is the process of creating software systems from existing software rather than building them from scratch [21]. 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. Research by Sojer and Henkel highlights the quantitative evidence, drivers, and impediments of code reuse in open source software development, further emphasizing its significance and challenges [23].

The study by Xu et al. [29] examines why developers choose to reuse existing libraries or re-implement similar functionality from scratch. The findings highlight

the benefits of library reuse, such as saving development time and effort, as well as the challenges, including compatibility issues and lack of documentation. These findings are highly relevant to understanding dependency management practices in the NPM ecosystem and complement the analysis presented in this thesis

2.3 Dependency and Dependent Library

The terms “dependency” and “dependent” describe the relationship between two libraries in software development. A dependency is a library, module, or package that a software project relies on to function correctly. Conversely, a dependent library is one that relies on one or more other libraries (dependencies) to operate.

Developers use dependencies to leverage existing features and avoid building functionality from scratch. However, maintaining and managing dependencies is crucial for ensuring the project’s continued functionality. This maintenance involves tasks such as updating, removing, and replacing dependencies to keep the software project up to date and free from issues related to dependencies.

2.4 Dependency Changes and Updates

Dependency changes and updates are a common occurrence in software development and can sometimes introduce problems for dependent libraries. These phenomenon are parts of the ecosystem evolution [15], reflecting the dynamic nature of software development where libraries continuously adapt and improve.

Breakage in dependent libraries can occur due to various reasons [4, 20, 25], including: feature modifications, incompatible provider versions, changes in object types, undefined objects, incorrect code semantics, failed provider updates, function renaming, or missing files. These issues highlight the fragility and inter-dependencies within software ecosystems, where a single change can have widespread effects.

The evolution of the ecosystem is driven by the need to enhance functionality, improve performance, and address security vulnerabilities. Studying this evolution helps us understand the growth rate of the ecosystem and whether developers continue to contribute to it [28], including the changes in dependencies. As de-

dependencies evolve, they often introduce new features, deprecate old ones, and fix bugs, necessitating updates in dependent libraries to maintain compatibility and leverage improvements. However, issues related to dependency updates can affect the ecosystem, leading to potential problems in dependent projects [9].

Dependency change issues extend beyond the NPM ecosystem. For example, a study examining deprecation in Javadoc [22] involved the manual analysis of 374 deprecated methods across four major Java APIs to determine whether deprecation reasons were documented.

2.5 Dependency Network

Dependency network is formed by the abundance to dependencies usage. A study of a topology of package dependency network reveal that the structure in NPM ecosystem is highly dependency-based. Because JavaScript lacks a proper standard library [8]. Many popular libraries are small and designed for compensate for the missing basic functionality that standard libraries would typically provide.

There was the interview of two NPM and five CRAN libraries maintainers [3] about dependencies management. NPM developers said it was too much information to follow the dependency changes through the network. R developers said that they prefer to limit the amount of dependency by limit the functionality or even absorb the code from dependencies.

2.6 Version Control System

Version control systems (VCS) are important tools in software development. They help developers manage changes to source code over time, allowing multiple people to work on the same project, track changes, and revert to earlier versions if needed. The most popular version control systems include GitHub, GitLab, and Bitbucket.

GitHub* is widely used in the open-source community for hosting and reviewing code, managing projects, and enabling collaboration among developers. GitLab[†]

*<https://github.com/>

[†]<https://about.gitlab.com/>

and Bitbucket[‡] are also popular platforms that offer similar features and serve many users. These platforms host many projects and dependencies, which are crucial for the development and maintenance of software libraries.

Understanding the role of version control systems provides context for this study, which focuses on libraries hosted on GitHub. By examining the reasons for dependency removal in these libraries, this research aims to provide insights into dependency management practices.

In VCS, a git commit is a snapshot of changes made to the code base. Each commit records the state of the repository at a specific point in time, allowing developers to track progress and revert to previous versions if necessary. Issues are used to track tasks, enhancements, and bugs in the project. They provide a way to document and discuss the work that needs to be done. Pull requests are a mechanism for submitting contributions to a project. A pull request allows developers to review changes before merging them into the main codebase, ensuring code quality and facilitating collaborative development.

This thesis is building on my previous research that focused on self-contained libraries. My earlier study investigated the characteristics and reasons of being self-contained library through dependency removal [13]. Previous studies, such as the one by Chuang et al., have concentrated on the process of removing dependencies [7]. Increasing dependency removal can help mitigate potential risks and disadvantages associated with dependencies.

[‡]<https://bitbucket.org/product/>

3 Dataset Preparation

This chapter explain the overview process of preparing the dataset. Figure 3.1 provides a visual overview of the library selection process for the dataset. The process starts by selecting the top 500 libraries with the highest number of dependents from libraries.io^{*}, sorting them in descending order by dependent count. Next, I systematically analyze the dependency tree of each library by using the Open Source Repository and Dependency Metadata from libraries.io. This analysis involves traversing through the dependency trees of all 500 libraries. A dependency tree represents the hierarchical structure of a library's dependencies, including both direct and transitive dependencies. After completing the traversal of these 500 dependency trees, I collected the unique libraries from each tree, resulting in a total of 2,763 unique libraries. The dataset is available in Zenodo, at <https://doi.org/10.5281/zenodo.12730734> [12]

^{*}<https://libraries.io/>

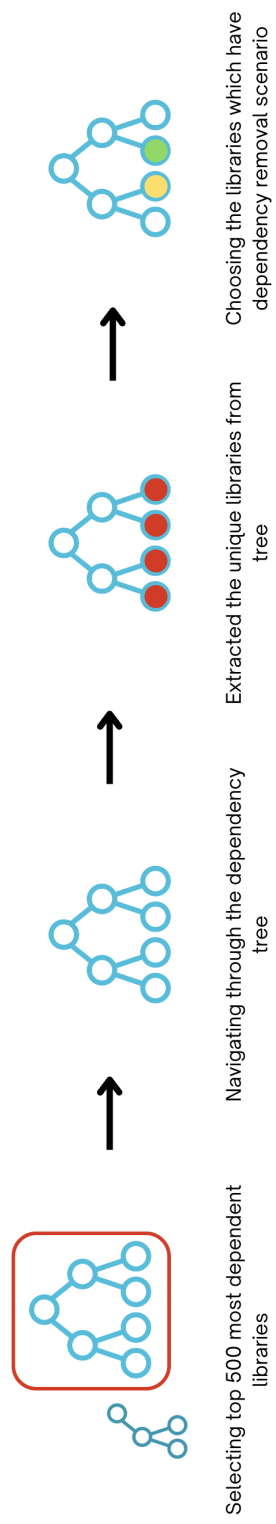


Figure 3.1: Overview of library selection

4 Empirical Study

In this chapter, I outline our approach to libraries which contained at least a dependency removal, with a focus on understanding the motivations for removing dependencies. This 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?

I investigated the dependency histories of transitional libraries, utilizing the registry npmjs API* to collect historical dependency data which not later than December 31, 2019. I 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 and developers can decide whether or not to install it. To maintain clarity in this investigation, I 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?

I conducted a detailed analysis of what removed dependencies from the identified libraries. Utilizing the dependency history data provided by the registry.npmjs

*<https://registry.npmjs.org>

API. By sorting the version history and comparing dependency lists across versions. My focus was specifically on versions where dependencies were reduced. I collected the removed dependencies and sort them by frequency of their removal.

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

Figure 4.1 provides a visual overview of gathering the Git commits and corresponding pull requests of the libraries. Start with examined the GitHub repository of each library to pinpoint the version where the maintainer reduced the target dependency. I collected relevant commits within that period. By matching pull requests with corresponding commits, I analyzed title, and description within the pull requests and the associated code changes. My goal was to identify the primary motivations driving the maintainer's decision to remove the target dependency. Therefore, I extracted the reasons for dependency removal from the analyze and grouped them into each category.

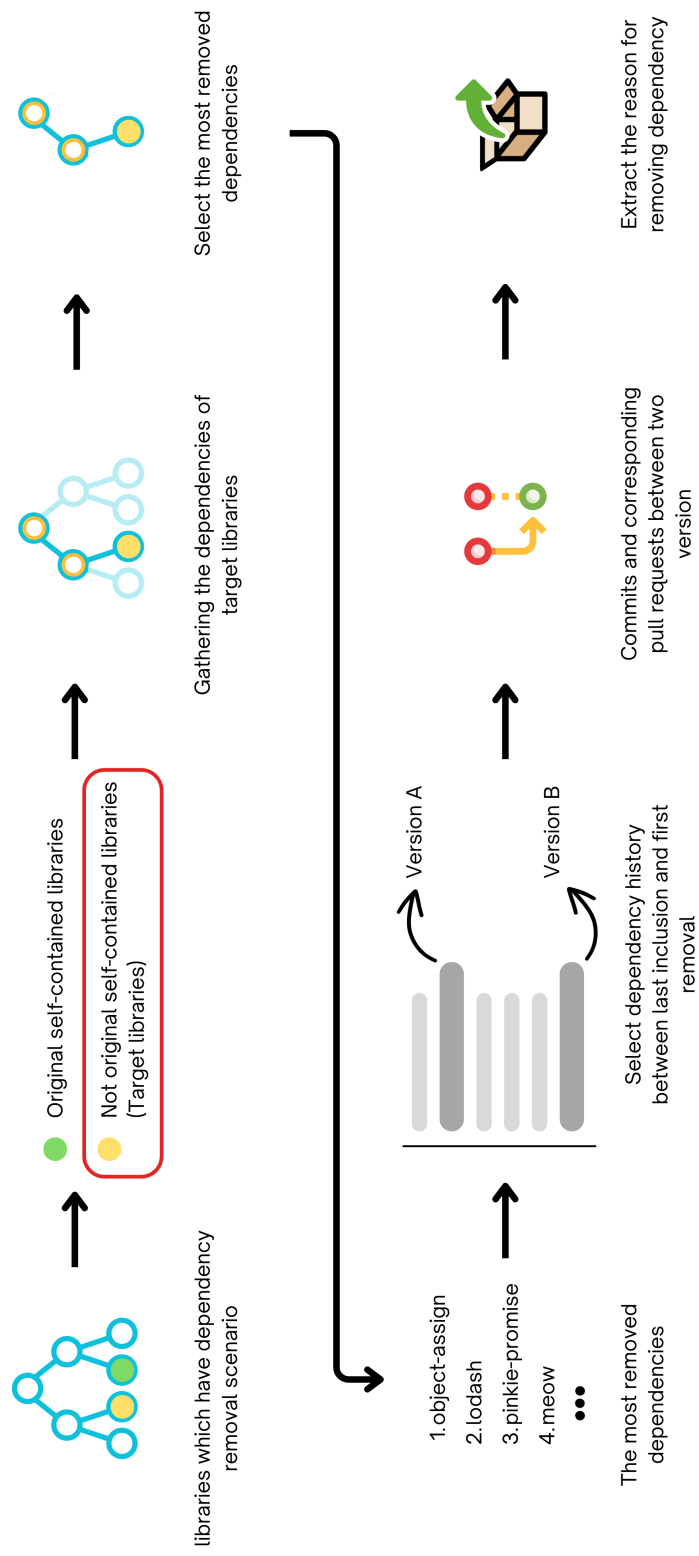


Figure 4.1: Overview of extracting the reasons for dependency removal

5 Result & Discussion

5.1 RQ 1: Prevalence

Table 5.1: Statistic of the libraries and dependencies.

Categories of library	No. libraries (%)	
Based libraries	2,763	(100%)
- Have dependency history before 2019	2,258	(81.72%)
- Have reduced dependencies	1,095	(39.63%)
- Can be gathered commits	841	(30.44%)

Our initial analysis of the top 500 most dependent libraries identified 2,763 unique libraries within the dependency chain. I examined the dependency history of each library and selected 2,258 libraries that had dependencies before December 31, 2019. Further investigation showed that 1,095 of these libraries had instances of dependency reduction. After gathering the relevant commits, I found that 841 libraries had commits that could be collected and analyzed. This thorough process allowed us to focus on libraries with clear instances of dependency removal, providing a solid dataset for our study.

Finding of RQ 1

Found 841 libraries or 30.44% of unique libraries are transitional libraries.

Table 5.2: Number of the gathered commits.

Types of Commit	No. commits (%)
All commits (not include code changed)	52,115 (100%)
Full commits (include code changed and obtainable)	36,131 (69.33%)
- Contain the package.json as a changed file	5,971 (11.46%)
- Related to dependency removal	1,551 (2.98%)
- After remove merge commits and false positive	682 (1.31%)

Table 5.3: Top 3 most removed dependencies.

Dependency's name	No. scenarios (%)
Top 3 most removed dependencies	80 (7.41%)
- object-assign	59 (5.46%)
- lodash	12 (1.11%)
- pinkie-promise	9 (0.83%)
All removing scenarios	1,080 (100%)

5.2 RQ 2: Most Reduced Libraries

From the 841 libraries that contain at least one dependency removal scenario, I examined the dependency history of each library. I identified the first version before and after the dependency removal scenario and gathered the commits made between these two versions. Table 5.2 shows the number of commits collected from all libraries. In total, I gathered 52,115 commits between the two versions across all libraries. Initially, these commits did not include code change details. I then gathered the full commit data, including code changes, but some commits could not be downloaded. As a result, the number of commits with code changes reduced to 36,131 commits. I filtered these to include only commits that listed `package.json` as a changed file and were related to dependency removal, yielding 1,551 commits. Finally, I removed false positives which is a refactor commit but the remaining dependencies are same with the code before the refactor commit and merge commits, resulting in 682 relevant commits.

From the 682 commits, I identified 820 dependencies that were removed. The most frequently removed dependencies were `object-assign`, `lodash`, and `pinkie-promise`. Figure 5.1 reveals a significant trend in dependency reduction, highlighting the most commonly removed dependencies.

Table 5.3 provides a detailed statistical breakdown of these libraries and their dependencies.

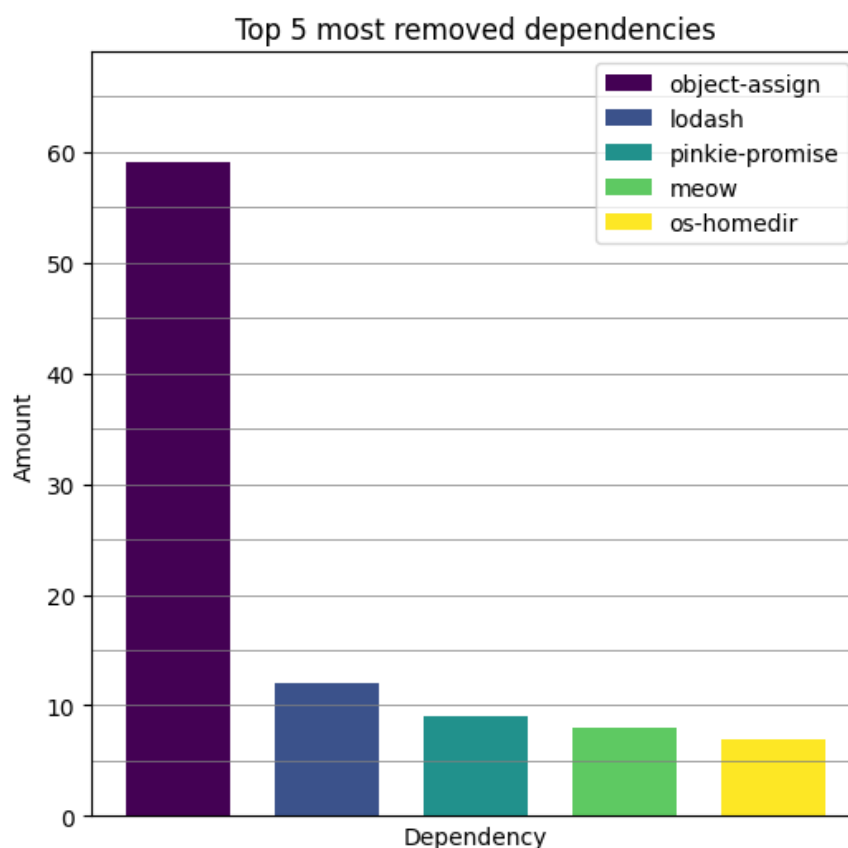


Figure 5.1: Top 10 most reduced dependencies

Finding of RQ 2

The most reduced dependency is `object-assign` which have been removed 59 times. Follow by `lodash` and `pinkie-promise` which have been removed 12 and 9 times.

Table 5.4: Terminology of classifying reasons from all dependency removal scenarios.

Terminology	Definition	Libraries (%)
Replace dependency with functions	Remove dependency and replace with built-ins or developer's custom function.	229 (21.26%)
Replace dependency with another dependency	Remove dependency and replace with another dependency.	224 (20.80%)
Remove bloat dependency	Remove dependency without any code changed.	203 (18.85%)
Shrink library	Remove dependency and remove function from dependency (reduce feature).	160 (14.86%)
Move dependencies to other field.	Move dependencies to devDependencies and peerDependencies	138 (12.81%)
Unknown	Cannot determine the reason. Need more detail.	123 (11.42%)
All		1,077 (100%)

5.3 RQ 3: Reasons for Reducing

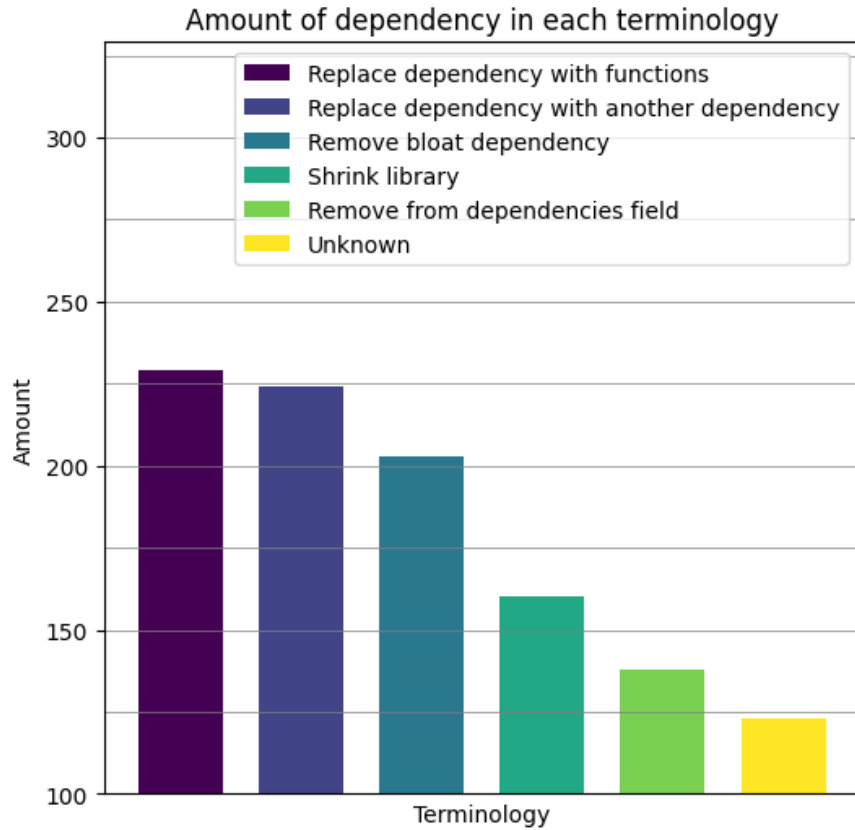


Figure 5.2: Types of dependency removal scenario

Guided by the findings of RQ 2, I classified 682 relevant commits, involving 1,077 instances of dependency removal, with each scenario representing the removal of one dependency from a dependent library. I matched the corresponding pull request to each commit to gather more detailed information. I then extracted the commit message and code changes from each commit, along with the title and description of the corresponding pull request.

In some commits, the dependency change may involve more than one dependency or include multiple dependency removal scenarios. Therefore, I classified each dependency removal scenario individually, as each dependency may be changed for different reasons.

Dependency	No. scenario
object-assign	43
pinkie-promise	9
os-homedir	7
isarray	3
kind-of	3

Table 5.5: Top 5 dependencies replaced by built-ins or custom functions

Table 5.4 and figure 5.2 presents the main terminology and definitions based on code changes, as well as the number of scenarios found for each term. These terms are used to classify the reasons for dependency removal, and the table also shows the overall number of libraries and commits associated with each classification based on the relevant commits gathered from each library.

According to Table 5.4, the most common reason for dependency removal is replacing a dependency with functions, found in 229 scenarios or 21.26% of all scenarios. This is followed by replacing a dependency with another dependency, found in 224 scenarios or 20.80% of all scenarios.

5.3.1 Terminology

Replace dependency with functions

This terminology means replace the dependency with built-ins functions or developer’s custom functions. Most of dependencies in this group are the micro dependencies which provide the replacement of basic functions in the JavaScript environment.

Table 5.5 lists the top 5 dependencies most frequently replaced by built-in functions or custom code. Table 5.8 compares the usage of these dependencies with their equivalent built-in functions or custom implementations.

The most frequently removed dependency in this category is `object-assign`. This dependency is often replaced with the built-in function `Object.assign`, which was introduced in Node.js version 4. Maintainers prefer to use `Object.assign` instead of `object-assign`. The next most commonly removed dependen-

cies are `pinkie-promise` and `os-homedir`. The `pinkie-promise` dependency is replaced by the built-in `Promise()` class. Meanwhile, `os-homedir` provides a function to identify the home directory of the user.

Replace dependency with another dependency

Dependencies in this category are those replaced by other dependencies. I identified two major types within this terminology:

First, there are dependencies that are sub-dependencies of the same bundled dependency, such as `lodash`. Table 5.6 lists the dependencies that are sub-dependencies of `lodash`. Developers can choose to install the entire bundled dependency with `npm install lodash` or install specific sub-dependencies like `lodash.isplainobject`. However, installing multiple sub-dependencies can lead to overlapping dependencies, which can increase the library size and the effort required to maintain these sub-dependencies.

I found some of the discussion about the overwhelming of dependencies and decided to replace it with `lodash`.*

In my view, one of the flaws in ESLint setup is the use of "modular" `lodash` NPM packages.

You could get rid of these dependencies:

```
"es6-map": "^0.1.3",
"escape-string-regexp": "^1.0.2",
"handlebars": "^4.0.5",
"lodash.clonedeep": "^3.0.1",
"lodash.merge": "^3.3.2",
"lodash.omit": "^3.1.0",
"object-assign": "^4.0.1",
"xml-escape": "~1.0.0"
```

by simply adding `lodash@4` that has all of those features (and no sub-dependencies. The sub-dependency tree of these dependencies is huge).

*<https://github.com/eslint/eslint/issues/5000#issuecomment-172909102>

And there is no reason whatsoever to use "modular" lodash packages (given that ESLint is a development tool).

Second, there are dependencies designed for similar tasks, such as `base64url`[†] and `base64-url`[‡]. Both `base64url` and `base64-url` are used for encoding and decoding in base64. Sometimes, developers might use both dependencies and later realize that, despite being different dependencies, their source code and functionality are similar. In such cases, they may prefer to reduce duplicate dependencies.

Remove bloat dependency

This terminology refers to the removal of dependencies from the dependencies fields without replacing them with another dependency, built-in functions, or custom functions. Dependencies in this category are considered unnecessary for the dependent libraries.

Shrink library

This category is similar to the “removing bloat dependency” type. However, the dependency removal scenarios in this category are not only related to removing dependencies but also involve removing the relevant code that relies on the features or functions of the removed dependencies. I grouped the dependencies in this category into two major subcategories based on the dependency removal scenarios:

1. **Reducing the features in the dependent libraries:** In this scenario, the maintainers of the dependent library choose to reduce the features of their library or remove code from their library without replacing it with any alternative or similar purpose code.
2. **Splitting the features into independent libraries:** In this scenario, the maintainers of the dependent library choose to split some features of their library into separate independent libraries.

[†]<https://www.npmjs.com/package/base64url>

[‡]<https://www.npmjs.com/package/base64-url>

Remove from dependencies field

Scenarios of this type are straightforward. The `package.json` file contains the description of the entire library, including the list of dependencies. There are various types of dependencies, such as `devDependencies` and `peerDependencies`. In these scenarios, the maintainers of the dependent library simply move dependencies from the “dependencies” field to other types of dependency fields without making any code changes.

Typically, the most common scenario in this category involves moving development dependencies from the `dependencies` section to the `devDependencies` section in `package.json`. Development dependencies are software library essential for the development process of an application but are not needed when the application is running in production. Their primary purpose is to assist with development tasks such as testing, building, linting, and formatting code.

Unknown

The reason for dependency removal in this terminology cannot be determined. In some commit the code changed and the commit description is not related.

5.3.2 Discussion

After a comprehensive analysis of commits and pull requests, and extracting the reasons for dependency removal, several interesting patterns emerged. This section discusses the findings from the previous section.

Retaining Features

From the classification of the reasons for dependency removal, in terms of retaining features can be distinguished into two groups: those that retain features and those that do not. Figure 5.3, shows that the majority of dependency removal scenarios involve removing dependency from the library while still retaining its features, with 591 scenarios falling into this category. In contrast, only 160 scenarios do not retain the features.

The group of retaining feature contains three categories: replace the dependency with functions, replace the dependency with another dependency, and



Figure 5.3: The Amount of Scenario Based on Retaining Features

remove from dependencies field. The first two categories involve replacing the dependency with alternative code, such as built-in functions or custom functions. The category "remove from dependencies field" indicates that the features from dependencies remain, but maintainers of the dependent library have moved the dependencies to other fields (e.g., `devDependencies` and `peerDependencies`) because there is no longer needed to use for the built library.

Meanwhile, the group not retaining features contains only the category “shrink library”. In this scenario, maintainers decided to remove the source code that relied on the dependencies, thus reducing the size and complexity of the library.

Finding of RQ 3

Most developers prefer to retain the existing features of their libraries rather than reduce or change them. The primary reason for removing a dependency is to replace it with functions, which affects more than 21% of the dependent libraries, followed by replacing the dependency with another dependency, affecting over 20% of the dependent libraries.

Sub-dependencies of lodash	No. scenario
lodash.isplainobject	4
lodash.clonedeep	3
lodash.debounce	2
lodash.isequal	2
lodash.flatten	2
lodash.foreach	2
lodash.map	2
lodash.assignwith	2
lodash.isfunction	2
lodash.merge	1
lodash.omit	1
lodash.filter	1
lodash.initial	1
lodash.last	1
lodash.pick	1
lodash.sortby	1
lodash.isstring	1
lodash.mapvalues	1
lodash.defaults	1
lodash.reduce	1
lodash.defaultto	1
lodash.differencewith	1
lodash.mergewith	1
lodash.unionwith	1
All	36

Table 5.6: List of sub-dependency of lodash

babel sub-dependencies	No. scenario
babel-core	1
babel-code-frame	1
babel-jest	1
babel-plugin-module-resolver	1
babel-plugin-preval	1
@babel/core	1
@babel/plugin-proposal-class-properties	1
@babel/plugin-transform-modules-commonjs	1
@babel/plugin-transform-runtime	1
@babel/plugin-transform-strict-mode	1
@babel/preset-env	1
@babel/preset-typescript	1
@babel/polyfill	1
@babel/runtime	1

Table 5.7: The list of babel sub-dependencies

Name of dependency	Usage of dependency	Replace built-ins functions or equivalent custom function
object-assign ^a	<code>var assign = require("object-assign"); assign({}, config, query);</code>	<code>Object.assign({}, config, query);</code>
pinkie-promise ^b	<code>var Promise = require("pinkie-promise"); then use Promise()</code>	Use <code>Promise()</code> without import any dependency
os-homedir ^c	<code>var osHomedir = require("os-homedir"); module.exports = osHomedir();</code>	<code>const os = require("os"); module.exports = os.homedir();</code>
isarray ^d	<code>var isarray = require("isarray"); isarray(s)</code>	<code>Array.isArray(s)</code>
kind-of ^e	<code>const kindof = require("kind-of"); kindOf(expectedCell) === "string"</code>	<code>typeof expectedCell === "string"</code>

Table 5.8: Example of Dependency with replace built-ins function or equivalent custom function

^a<https://github.com/webpack/loader-utils/commit/ae1d865dbc6b15ccfb4ee8fa6bf7e4fd6c05ea20>

^b<https://github.com/sindresorhus/de1/commit/614368bb1a213925e7e038650614ee385b7203be>

^c<https://github.com/sindresorhus/home-or-tmp/commit/6c288a8d1a3eb9019e2b78bb1aa95327cb7d8685>

^d<https://github.com/browserify/labeled-stream-splicer/commit/e05eeef9c025ae0b700a33e8cdd03e243222f911>

^e<https://github.com/cli-table3/commit/7d8618c3a2ec2603e188864f941fad6f37bc498e>

6 Implications

This thesis investigates the dependency removal in the NPM ecosystem, providing important insights into dependency management within modern software development. The findings of this study have several key implications for both developers and researchers:

- **Maintainers of Dependencies:** Common reasons for dependency removal serve as implicit feedback from developers. Maintainers can use this feedback to improve their libraries and provide better dependencies to others.
- **Software Developers:** Understanding the common reasons for dependency removal, such as replacing dependencies with built-in functions or custom code, helps developers make better decisions about managing their libraries. They can use these insights to choose dependencies more effectively, minimize external dependencies, reduce the risk of cascading issues, and improve overall software quality.
- **Tool Support:** The detailed analysis of commit messages, code changes, and pull requests provides a foundation for developing tools to automate the detection and management of dependencies. These tools can help developers maintain clean and efficient dependency trees.

7 Threats to Validity

7.1 Internal Validity

1. **Selection Bias:** The selection of the top 500 libraries based on the number of dependents may introduce bias, as these libraries might not represent the entire NPM ecosystem. This thesis gathered only the libraries hosted on GitHub, excluding those that may have many dependents but are located on other version control systems such as GitLab and Bitbucket. This limitation could affect the generalizability of the findings.
2. **Commit and Pull Requests Analysis:** The analysis relies on the accuracy and completeness of commit messages and pull request descriptions. GitHub allows users to revise commit messages after they have been pushed* and to change pull request titles and descriptions†. Maintainers can modify commit messages and pull request details at any time, which can affect the accuracy of the information. Inaccurate information could lead to incorrect interpretations of the reasons for dependency removal.
3. **Manual Classification:** The classification of dependency removal reasons involves a degree of subjectivity. Although efforts were made to minimize bias, personal judgment may influence the categorization process.

*<https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/changing-a-commit-message>

†<https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/changing-the-base-branch-of-a-pull-request>

7.2 External Validity

1. **Applicability:** The findings are based on a specific subset of NPM libraries and may not be applicable to other package managers or ecosystems. Further studies are needed to validate the results across different contexts.
2. **Time Frame:** The dataset only includes commits up to December 31, 2019. Trends in dependency management may have evolved since then, potentially limiting the applicability of the results to current practices.

7.3 Construct Validity

1. **Definition of Dependency Removal:** The definition of what constitutes a dependency removal scenario may vary. This study defines it based on specific criteria, which may not cover all possible scenarios of dependency management in practice.
2. **Measurement of Impact:** The impact of dependency removal on software quality and maintainability is inferred from qualitative analysis. Quantitative measures of impact, such as performance metrics or user satisfaction, are not included in this study.

7.4 Conclusion Validity

1. **Data Accuracy:** The reliability of this analysis heavily depends on the accuracy of the dataset obtained from libraries.io, registry.npmjs, and GitHub repositories. Libraries.io provided the ranking of the top 500 most dependent libraries. Registry.npmjs supplied the information and metadata of all dependencies in the dataset, and GitHub was the source for gathering commits and pull requests. All data was collected in 2023. Any inaccuracies in these data sources could affect the validity of the conclusions drawn.
2. **Causal Relationships:** While correlations between dependency removal and various factors are identified, establishing causal relationships requires further investigation. The study provides insights but does not definitively

determine causality. For instance, the category “Remove bloat dependency” mentioned in Section 5.3 reports that dependencies have been removed from the dependent libraries. However, the actual reason for dependency removal might not be “Remove bloat dependency” as the root cause could be found in other commits that are not directly related to dependency removal. This is because the criteria for dependency removal are based on the list of dependencies in the `package.json` file. Further investigation is needed to accurately distinguish the reasons for dependency removal.

By acknowledging these threats to validity, this thesis provides a clear and honest analysis of dependency removal practices in the NPM ecosystem. Recognizing these limitations helps in accurately interpreting the findings and highlights areas for future research. This transparency ensures that the conclusions are understood within the context of these potential limitations and encourages further study and improvement of dependency management practices in software development.

8 Conclusion

This thesis has addressed the crucial topic of dependency management within the NPM ecosystem by investigating the scenarios and reasons for dependency removal. By analyzing a substantial dataset of 52,115 commits from 2,763 unique libraries, I identified significant patterns and trends in how and why dependencies are removed.

- **Developers Prefer to Retain Features:** Many categories indicate that developers prefer to retain features in their libraries. This preference is evident from the various categories identified in the study.
- **Most replacements are made with built-in functions:** Using built-in functions is faster and easier compared to implementing custom functions. However, in some cases, developers incorporate the source code from the dependency directly into their library, which is similar to using built-in functions.

The methodology of matching commits to corresponding pull requests provided deeper insights into the motivations behind these changes. Common reasons included simplifying the code base, reducing library size, and minimizing maintenance efforts. These insights offer practical guidance for developers on best practices in dependency management.

Moreover, the thesis proposes future research directions, such as developing a comprehensive taxonomy of dependency removal reasons and creating mated tools for dependency analysis. These tools would support developers in maintaining clean and efficient project dependencies, enhancing overall software quality.

In summary, this thesis makes a significant contribution to software engineering by elucidating the practices and rationales of dependency removal in the NPM

ecosystem. The findings and recommendations presented provide valuable guidance for developers, researchers, and educators, aiming to foster more robust and maintainable software systems.

9 Future Work

Based on the studies conducted in this thesis, I have completed two major investigations:

1. **A Preliminary Study on Self-Contained Libraries in the NPM Ecosystem (previous research paper):** This study reveals the characteristics of self-contained libraries and investigates the reasons behind their self-contained nature.
2. **An Empirical Study on Dependency Removal in the NPM Ecosystem (this thesis):** In this study, I expanded the scope to include dependencies that have undergone removal scenarios. This resulted in the identification of various categories of reasons for dependency removal.

To build on the findings of this thesis, I plan to undertake more further steps: developing a taxonomy of reasons for dependency removal and creating tools to detect code that relies on specific dependencies within a project. Below are the details of these plans:

- **Taxonomy of Reasons for Dependency Removal:** The first step after completing this thesis is to develop a comprehensive taxonomy of the reasons for dependency removal. This taxonomy will categorize similar reasons together and show the frequency of dependencies in each group. This can help developers better understand the rationale behind choosing to install or remove dependencies. To create this taxonomy, I will need to expand the dataset beyond the 500 most dependent libraries to cover a broader scope of the ecosystem.
- **Tool to Detect Code Relying on Specific Dependencies:** Building on the taxonomy, I plan to develop a tool that helps detect code relying

on specific dependencies within a project. This tool will enhance usability by reporting which parts of the code depend on each dependency and highlighting the consequences of removing those dependencies. Additionally, the tool will provide recommendations on which dependencies are most likely to be removed by other developers, along with the reasons for their removal. The purpose of this tool is to assist developers in minimizing dependencies and making their libraries more efficient.

By pursuing these future research directions, they will advance the understanding of dependency management and provide practical tool to support developers in creating more maintainable and efficient software systems.

Acknowledgement

I would like to express my sincere gratitude to the following individuals. Completing my Master's degree and this thesis would have been impossible without their support and encouragement.

First and foremost, I extend my deepest thanks to my supervisor, Professor Kenichi Matsumoto, for giving me the valuable opportunity to join the software engineering laboratory and pursue my Master's degree. His guidance and encouragement throughout my studies have been invaluable.

I am also grateful to my committee members, Professor Keiichi Yasumoto, Professor Takashi Ishio, Associate Professor Raula Gaikovina Kula, and Assistant Professor Shimari Kazumasa, for their constructive feedback and advice, which have greatly improved the quality of this work.

Additionally, I owe special thanks to my co-supervisor, Associate Professor Raula Gaikovina Kula for providing me with the opportunity to attend an international conference.

I would also like to thank Brittany Reid, a post-doc in my laboratory, for her support and advice, which have been essential in completing my research and thesis.

Special thanks to my colleagues and lab mates for their constant support, including emotional support, fruitful discussions, and camaraderie throughout this journey. Their encouragement has been a source of motivation and inspiration.

I acknowledge the financial support provided by the Konosuke Matsushita Memorial Foundation (KMMF) *, which covered all expenses during my Master's studies in Japan, including tuition, enrollment fees, and monthly expenses. They

*<https://matsushita-konosuke-zaidan.or.jp/en/>

Bibliography

- [1] 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*, pages 385–395, 2017.
- [2] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering*, 25:1168–1204, 2020.
- [3] C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 86–89. IEEE, 2015.
- [4] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. 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*, pages 109–120, 2016.
- [5] X. Chen, R. Abdalkareem, S. Mujahid, E. Shihab, and X. Xia. Helping or not helping? why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering*, 26:1–24, 2021.
- [6] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, 26:1–28, 2021.
- [7] C.-C. Chuang, L. Cruz, R. van Dalen, V. Mikovski, and A. 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. doi: 10.1109/SCAM55253.2022.00017.

- [8] A. Decan, T. Mens, and M. Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proceedings of the 10th european conference on software architecture workshops*, pages 1–4, 2016.
- [9] A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 2–12. IEEE, 2017.
- [10] A. Decan, T. Mens, and P. Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.
- [11] F. Hou and S. Jansen. A systematic literature review on trust in the software ecosystem. *Empirical Software Engineering*, 28(1):8, 2023.
- [12] P. Jaisri. The commit history of the dependent libraries and the associated pull requests related to dependency removal. <https://doi.org/10.5281/zenodo.12730734>, 2024.
- [13] P. Jaisri, B. Reid, and R. G. Kula. A preliminary study on self-contained libraries in the npm ecosystem. *arXiv preprint arXiv:2406.11363*, 2024.
- [14] S. Jansen and M. A. Cusumano. Defining software ecosystems: a survey of software platforms and business network governance. In *Software ecosystems*, pages 13–28. Edward Elgar Publishing, 2013.
- [15] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017.

- [16] 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. *arXiv preprint arXiv:1709.04638*, 2017.
- [17] T. Mens. An ecosystemic and socio-technical view on software maintenance and evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–8. IEEE, 2016.
- [18] I. Pashchenko, D.-L. Vu, and F. Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1513–1531, 2020.
- [19] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo. Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26:1–34, 2021.
- [20] S. Raemaekers, A. Van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, 2014.
- [21] J. Sametinger. *Software engineering with reusable components*. Springer Science & Business Media, 1997.
- [22] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. 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.
- [23] M. Sojer and J. Henkel. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems*, 11(12):868–901, 2010.
- [24] C. Soto-Valero, T. Durieux, and B. Baudry. A longitudinal analysis of bloated java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1021–1031, 2021.

- [25] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. 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.
- [26] C. Wang, R. Wu, H. Song, J. Shu, and G. Li. smartpip: A smart approach to resolving python dependency conflict issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.
- [27] S. Wattanakriengkrai, D. Wang, R. G. Kula, C. Treude, P. Thongtanunam, T. Ishio, and K. Matsumoto. Giving back: Contributions congruent to library dependency changes in a software ecosystem. *IEEE Transactions on Software Engineering*, 49(4):2566–2579, 2022.
- [28] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th international conference on mining software repositories*, pages 351–361, 2016.
- [29] B. Xu, L. An, F. Thung, F. Khomh, and D. Lo. Why reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*, 25:755–789, 2020.
- [30] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security symposium (USENIX security 19)*, pages 995–1010, 2019.

Publication List

1. Pongchai Jaisri, Brittany Reid, and Raula Gaikovina Kula. A Preliminary Study on Self-Contained Libraries in the NPM Ecosystem. To be published at the Springer's Studies in Computational Intelligence Series (SCIS), 2024.