

# **Effective Dependency Management for the JavaScript Software Ecosystem**

**Suhaib Mujahid**

**A Thesis**

**in**

**The Department**

**of**

**Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements**

**for the Degree of**

**Doctor of Philosophy (Software Engineering) at**

**Concordia University**

**Montréal, Québec, Canada**

**October 2021**

**© Suhaib Mujahid, 2021**

CONCORDIA UNIVERSITY  
School of Graduate Studies

This is to certify that the thesis prepared

By: **Suhaib Mujahid**  
Entitled: **Effective Dependency Management for the JavaScript Software  
Ecosystem**

and submitted in partial fulfillment of the requirements for the degree of

**Doctor of Philosophy (Software Engineering)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

\_\_\_\_\_  
*Dr. Fuzhan Nasiri* Chair

\_\_\_\_\_  
*Dr. Audris Mockus* External Examiner

\_\_\_\_\_  
*Dr. Yann-Gaël Guéhéneuc* Examiner

\_\_\_\_\_  
*Dr. Tse-Hsun Chen* Examiner

\_\_\_\_\_  
*Dr. Jamal Bentahar* Examiner

\_\_\_\_\_  
*Dr. Emad Shihab* Supervisor

Approved by

\_\_\_\_\_  
Dr. Lata Narayanan, Chair  
Department of Computer Science and Software Engineering

22 November 2021

\_\_\_\_\_  
Dr. Mourad Debbabi, Dean  
Gina Cody School of Engineering and Computer Science

# Abstract

## Effective Dependency Management for the JavaScript Software Ecosystem

**Suhaib Mujahid, Ph.D.**

**Concordia University, 2021**

Open source software ecosystems are essential to software development. Developers depend on packages from the ecosystems to utilize their functionalities and avoid having to reinvent the wheel. On the one hand, this allows developers to write less code, increasing productivity, improving quality, and delivering more features. On the other hand, the package dependencies themselves must be maintained. The overhead starts with the process of selecting a quality package to use out of a large set of packages, going through updating the dependencies and avoiding breakage-inducing versions, ending with replacing obsolete dependencies and finding better alternatives. Neglecting the maintenance of the dependencies can have an expensive negative impact on the software quality. Hence, in this thesis, we propose facilitating dependency management activities, encouraging developers to keep healthy dependencies in their projects.

We employ information extracted from the software ecosystem to help developers better manage their software dependencies. We first present an empirical study on the factors used by developers to select dependency packages from the npm software ecosystem. Next, we propose an approach that leverages tests from the ecosystems to help identify breakage-inducing versions, which increase developers' confidence in updating the dependencies and help them to make more informed decisions when they update dependencies. Also, we propose an approach to identify packages in decline as early as possible. The underlying

rationale of our approach is that the decline in community interest leads to having packages used less over time, becoming less frequently maintained, and eventually, could become abandoned. Furthermore, we propose an approach to find alternatives to replace packages in decline. Finally, we empirically evaluated our approach and characterized the alternative packages.

# Related Publications

Earlier versions of the work presented in this thesis has been previously published or submitted to the venues listed below:

- **S. Mujahid**, R. Abdalkareem, E. Shihab, and S. McIntosh. “Using others’ tests to identify breaking updates.” In Proceedings of the 17th International Conference on Mining Software Repositories (MSR), pages 466–476, [2020](#).
- **S. Mujahid**, D. Costa, R. Abdalkareem, E. Shihab, MA. Saied, and B. Adams. “Towards Using Package Centrality Trend to Identify Packages in Decline.” IEEE Transactions on Engineering Management Journal (TEM), accepted, 15 pages, [2021](#).
- **S. Mujahid**, R. Abdalkareem, and E. Shihab. “What are the characteristics of highly-used packages? A case study on the npm ecosystem.” IEEE Transactions on Software Engineering Journal (TSE), under review, 14 pages.
- **S. Mujahid**, D. Costa, R. Abdalkareem, and E. Shihab. “Where to Go Now? Finding Alternatives for Declining Packages in the npm Ecosystem.” IEEE Transactions on Software Engineering Journal (TSE), under review, 13 pages.

In addition, the following publications are indirectly related to the material in this thesis, and were produced in parallel to the research performed for this thesis:

- G. Cabral, L. Minku, E. Shihab, **S. Mujahid**. “A Fine-Grained Investigation of Class Imbalance Evolution in Just-in-Time Software Defect Prediction.” In Proceedings

of the 41st ACM/IEEE International Conference on Software Engineering (ICSE), pages 666-676, [2019](#).

- R. Abdalkareem, **S. Mujahid**, E. Shihab, and J. Rilling. “Which Commits Can Be CI Skipped?” In IEEE Transactions on Software Engineering Journal (TSE), pages 448-463, [2019](#).
- R. Abdalkareem, V. Oda, **S. Mujahid**, and E. Shihab. “On the Impact of Using Trivial Packages: An Empirical Case Study on npm and PyPI.” In Empirical Software Engineering Journal (EMSE), pages 1168-1204, [2020](#).
- R. Abdalkareem, **S. Mujahid**, and E. Shihab. “A Machine Learning Approach to Improve the Detection of CI Skip Commits.” In IEEE Transactions on Software Engineering Journal (TSE), pages 448-463, [2020](#).
- J. Hoyos, R. Abdalkareem, **S. Mujahid**, E. Shihab and A. Espinosa. “On the Removal of Feature Toggles: A Study of Python Projects and Practitioners Motivations.” In Empirical Software Engineering Journal (EMSE), 26 pages, [2021](#).
- X. Chen, R. Abdalkareem, **S. Mujahid**, E. Shihab and X. Xia. “Helping or not Helping? Why and How Trivial Packages Impact the npm Ecosystem.” In Empirical Software Engineering Journal (EMSE), 24 pages, [2021](#).
- D. Costa, **S. Mujahid**, R. Abdalkareem, and E. Shihab. “Breaking Type-Safety in Go: An Empirical Study on the Usage of the unsafe Package.” In IEEE Transactions on Software Engineering Journal (TSE), 17 pages, [2021](#).

# Acknowledgments

First and foremost, I would like to thank Allah for blessing me with this opportunity.

I would like to thank my supervisor Dr. Emad Shihab for his endless support and dedication during this journey, as well as his unparalleled motivation and immense knowledge, without which this thesis would not have been completed. I have learned a great deal from you, words could never be enough to express my gratitude. You played formative roles in my development as a researcher and as a person. Your unique personality, as a supervisor and a friend, has made this journey enjoyable.

I would also like to thank my committee members, Drs. Audris Mockus, Yann-Gaël Guéhéneuc, Tse-Hsun Chen, and Jamal Bentahar for taking time out to read my thesis. Their insightful comments have enhanced this thesis.

I have also had the privilege to collaborate and discuss my research with great researchers. I want to thank Drs. Rabe Abdalkareem, Diego Costa, Shane McIntosh, Bram Adams, and Mohamed Aymen Saied for sharings their insights and advices.

My appreciation extends to my fellow colleagues, Ahmad Abdellatif, Mahmoud Al-fadel, Mohamed Elshafei, SayedHassan Khatoonabadi, Khaled Badran, Nicholas Nagy, Abbas Javan, Juan Hoyos, Giancarlo Sierra, Jinfu Chen, Kundi Yao, Jasmine Latendresse, Hosein Nourani, Atique Reza, Xiaowei Chen, Mouafak Mkhallalati, Patrick Ayoup, and everyone else in the Data-driven Analysis of Software (DAS) Lab.

It is a pleasure to thank my friends who inspired and supported me tirelessly through thick and thin. A special thanks to Samer, Rame, Saif, Amro, Khalil, and Asem. You guys

always fueled me with utmost motivation. I am proud to call you my closest friends.

I thank my parents for always being there for me. The greatest motivation that makes me continue my path toward success is seeing joy and pride in their faces. I likewise thank my sisters for their unwavering moral support and encouragement. Lastly, I would like to thank my beloved wife Samiha, who has stood by my side throughout this journey.



# Dedication

*To my parents, sisters, and wife.*

# Contents

<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xviii</b>
<b>1 Introduction and Research Statement</b>	<b>1</b>
1.1 Research Statement . . . . .	3
1.2 Thesis Overview . . . . .	4
1.3 Thesis Contributions . . . . .	8
1.4 Thesis Organization . . . . .	8
<b>2 Background</b>	<b>9</b>
2.1 Software Ecosystems . . . . .	9
2.2 Package Dependencies . . . . .	10
2.2.1 Semantic Versioning . . . . .	10
2.2.2 Technical Lag . . . . .	11
2.2.3 Dependency Migration . . . . .	11
<b>3 Related Work</b>	<b>13</b>
3.1 Reusing Packages from Software Ecosystems . . . . .	13
3.2 Studying API Breakage Changes . . . . .	16
3.3 Studying Package Evolution in Software Ecosystems . . . . .	17

3.4	Recommending Package Dependencies . . . . .	19
<b>4</b>	<b>An Empirical Study on the Characteristics of Highly-Selected Packages</b>	<b>21</b>
4.1	Introduction . . . . .	21
4.2	Qualitative Analysis . . . . .	24
4.2.1	Study Design . . . . .	24
4.2.2	Study Results . . . . .	30
4.3	Quantitative Analysis . . . . .	34
4.3.1	Study Design . . . . .	35
4.3.2	Analysis Method . . . . .	39
4.3.3	Study Results . . . . .	43
4.3.4	Lifetime Analysis . . . . .	46
4.4	Discussion and Implications . . . . .	48
4.5	Threats to Validity . . . . .	49
4.5.1	Internal Validity . . . . .	50
4.5.2	Construct Validity . . . . .	50
4.5.3	External Validity . . . . .	51
4.6	Chapter Summary . . . . .	52
<b>5</b>	<b>An Approach to Identify Breaking Updates</b>	<b>53</b>
5.1	Introduction . . . . .	53
5.2	Background and Motivating Example . . . . .	56
5.3	Study Design . . . . .	58
5.3.1	Corpus of Candidate Packages . . . . .	58
5.3.2	Selection of Case Studies . . . . .	61
5.3.3	Detection of Breakage-Inducing Versions . . . . .	63
5.4	Case Study Results . . . . .	65

5.5	Discussion . . . . .	73
5.5.1	Technique Scalability . . . . .	73
5.5.2	Failed Builds . . . . .	75
5.6	Threats to Validity . . . . .	76
5.6.1	Threats to Internal Validity . . . . .	76
5.6.2	Threats to External Validity . . . . .	77
5.7	Chapter Summary . . . . .	78
<b>6</b>	<b>An Approach to Identify Packages in Decline</b>	<b>79</b>
6.1	Introduction . . . . .	79
6.2	Motivation Example . . . . .	82
6.3	Approach . . . . .	84
6.3.1	Calculating Centrality Trends . . . . .	85
6.3.2	Detecting Packages In Decline . . . . .	89
6.4	Evaluation Datasets . . . . .	91
6.4.1	Extracting <i>npm</i> s Validation Baseline Corpora . . . . .	91
6.4.2	Survey Validation Baseline Corpus . . . . .	94
6.4.3	Deprecated Packages Corpus . . . . .	95
6.5	Results . . . . .	96
6.6	Tool Prototype . . . . .	110
6.7	Threats to Validity . . . . .	112
6.7.1	Threats to Internal Validity . . . . .	112
6.7.2	Threats to External Validity . . . . .	113
6.8	Chapter Summary . . . . .	113
<b>7</b>	<b>An Approach to Find Alternative Packages</b>	<b>115</b>
7.1	Introduction . . . . .	115

7.2	Approach . . . . .	118
7.2.1	Detect Dependency Change Events . . . . .	119
7.2.2	Extract Dependency Migration Patterns . . . . .	120
7.2.3	Calculate Centrality Trends . . . . .	122
7.2.4	Select Package Alternatives . . . . .	122
7.2.5	Extract Pull Request Examples . . . . .	123
7.3	Accuracy of the Approach . . . . .	124
7.3.1	Generate Dependency Migration Suggestions . . . . .	124
7.3.2	Manual Evaluation Process . . . . .	125
7.3.3	Results . . . . .	126
7.4	Usefulness of the Approach . . . . .	127
7.4.1	Survey Design . . . . .	128
7.4.2	Participant Recruitment . . . . .	130
7.4.3	Survey Participants . . . . .	131
7.4.4	Results . . . . .	133
7.5	Characteristics of the Suggestions . . . . .	137
7.5.1	Manual Classification . . . . .	138
7.5.2	Results . . . . .	139
7.6	Threats to Validity . . . . .	142
7.6.1	Threats to Internal Validity . . . . .	142
7.6.2	Threats to External Validity . . . . .	143
7.7	Chapter Summary . . . . .	143
<b>8</b>	<b>Conclusions and Future Work</b>	<b>145</b>
8.1	Conclusion and Findings . . . . .	145
8.2	Limitations . . . . .	148
8.3	Future Work . . . . .	148

8.3.1	Replication on Other Software Ecosystems . . . . .	148
8.3.2	Extend Scope of Dependent Projects . . . . .	149
8.3.3	Investigate Why a Package Is in Decline . . . . .	150
8.3.4	Predict Future Central Packages . . . . .	150
8.3.5	Expand Dependency Migration Suggestions . . . . .	150
8.3.6	Automate Dependency Migrations . . . . .	151

<b>Bibliography</b>		<b>152</b>
---------------------	--	------------

# List of Figures

Figure 1.1	Dependency management activities and the studied topics in this thesis. . . . .	2
Figure 4.1	An overview of our study design. . . . .	23
Figure 4.2	Survey responses regarding how often our survey participants search for npm packages. . . . .	29
Figure 4.3	A histogram for the used npm packages. . . . .	37
Figure 4.4	The hierarchical clustering shows the factors that present highly-selected packages. . . . .	41
Figure 4.5	The nomogram visually presents the impact of each of the studied factors in determining highly-selected npm packages. . . . .	46
Figure 5.1	Motivating example overview and used terminology. . . . .	57
Figure 5.2	Data collection overview . . . . .	59
Figure 5.3	The approach overview. . . . .	63
Figure 5.4	The distribution of test coverage for the studied cases based on running the tests of their dependent projects. . . . .	66
Figure 5.5	The cumulative statement test coverage for the selected ten cases based on running the testes of their dependent projects. . . . .	68
Figure 5.6	The cumulative statement test coverage for the selected ten cases based on running the testes of their dependent projects companied with their own tests. . . . .	69

Figure 5.7	The distribution of the time that tests consume to pass or fail the builds. . . . .	74
Figure 5.8	Time in seconds to the cumulative percentage of dependent projects that completed their tests. . . . .	75
Figure 5.9	The classification of the failed builds . . . . .	76
Figure 6.1	Evolution of the Moment.js package on the centrality (PageRank), the number of downloads, and the number of dependents. . . . .	83
Figure 6.2	The approach to calculate centrality trends. . . . .	85
Figure 6.3	Illustration of our dependency graph build process . . . . .	85
Figure 6.4	Example of a package trend in decline. . . . .	90
Figure 6.5	Timeline used to select validation baseline from <i>npm</i> s. . . . .	92
Figure 6.6	ROC curve with the AUC value for the evaluation based on the <i>npm</i> s baseline. . . . .	98
Figure 6.7	Examples of packages that our approach only detected the decline after the <i>npm</i> s score. . . . .	99
Figure 6.8	Letter-value plots for the distribution of how early our approach can detect packages in decline. . . . .	104
Figure 6.9	The distribution of the correlation between centrality and the metrics. . . . .	106
Figure 6.10	Examples of packages with strong negative correlation between the centrality trend and the number of dependents trend. . . . .	107
Figure 6.11	Line plots showing the trend of centrality alongside with the trend of other metrics. . . . .	109
Figure 6.12	A screenshot of the npm website showing the package underscore with the integrated centrality information from our Chrome extension. . . . .	110
Figure 6.13	The time required to update the dependency graph and calculate the centrality for all packages. . . . .	111



Figure 7.1 Our approach to suggest package alternatives. . . . . 118

Figure 7.2 Survey responses regarding how often our survey participants search  
for package alternatives. . . . . 132

Figure 7.3 The awareness of participants about the suggested alternative pack-  
ages and the community’s migration trends. . . . . 134

Figure 7.4 Survey responses regarding the usefulness of having a tool that gen-  
erate alternative suggestions using our approach. . . . . 134

Figure 7.5 Survey responses regarding the support of migration their current  
projects to use the alternative packages. . . . . 136

# List of Tables

Table 4.1	List of factors used in selecting a packages from the npm ecosystem. .	26
Table 4.2	Participants’ position, experience in software development, JavaScript and usage of the npm package manager. . . . .	28
Table 4.3	Survey results of the factors used in selecting a package from the npm ecosystem. . . . .	31
Table 4.4	List of factors values with their description. . . . .	38
Table 4.5	The result of our logistic regression analysis for investigating the most important factors. . . . .	44
Table 4.6	Highly-selected Vs. Not highly-selected packages: Mann-Whitney test ( $p$ -value) and Cliff’s Delta ( $d$ ). . . . .	45
Table 4.7	The result of our logistic regression analysis for investigating the most important factors considering lifetime of the studied packages. . . . .	47
Table 5.1	The Selected Ten Downgrading Cases. . . . .	61
Table 5.2	Builds and Tests Summary. . . . .	70
Table 6.1	Results of using the centrality trend to classify packages from the <i>npm</i> s validation baseline. . . . .	98
Table 6.2	Results of three datasets on how early in months our approach can detect packages in decline. . . . .	103
Table 7.1	Summary of the suggested alternatives categorize by their functionalities. . . . .	126

Table 7.2	Questions in our survey about the alternative package suggestions. . .	129
Table 7.3	Participants' position and experience in software development, JavaScript development, and using npm. . . . .	132
Table 7.4	Participants' responses on how helpful are the examples of dependency migrations from other projects? . . . . .	135
Table 7.5	The activities of the pull requests that performed the dependency migrations. . . . .	139
Table 7.6	The motivations of 62 pull requests that performed the dependency migrations. . . . .	140

# Chapter 1

## Introduction and Research Statement

Software ecosystems are the backbone of modern software development. The increase in the popularity of open source software ecosystems has encouraged the reuse of third-party packages and turned it into prevailing practice. Nowadays, developers spend less time implementing common functionalities. Instead, they reuse functionalities from the software ecosystems, where other developers publish reusable code as packages. Thus, developers commonly publish their packages to the community (Wittern, Suter, & Rajagopalan, 2016), making an ecosystem such as Node.js Package Manager (npm)<sup>1</sup> the host of more than 1.7 million packages (DeBill, 2021).

Prior work shows that depending on third-party packages has many advantages. The advantages include allowing developers to build software systems faster, deliver richer features, and even achieve higher quality (Abdalkareem, Nourry, Wehaibi, Mujahid, & Shihab, 2017; Abdalkareem, Oda, et al., 2020). However, these advantages often come at an increased cost of selecting and managing the dependencies (Mirhosseini & Parnin, 2017). Given the massive number of packages from which to choose, selecting a suitable package can be challenging, especially considering that many packages provide the same functionality. The large size and rapid evolution of these ecosystems have other downsides as

---

<sup>1</sup><https://www.npmjs.com>

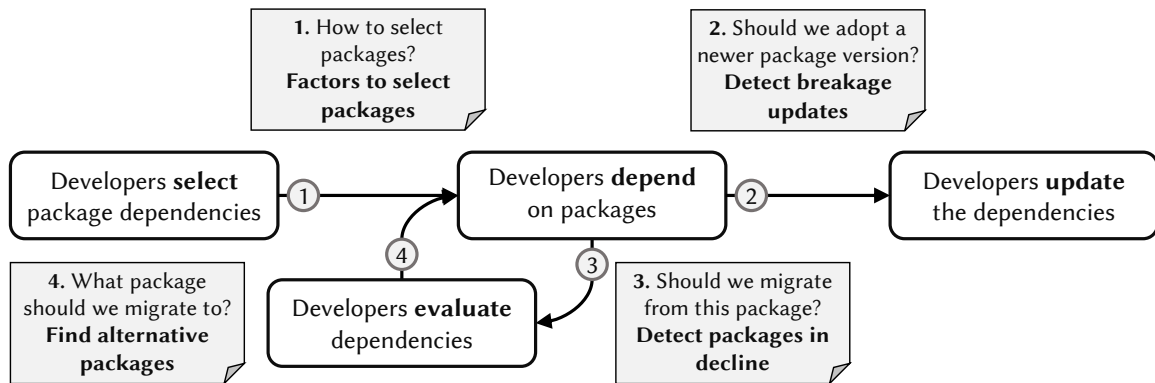


Figure 1.1: Dependency management activities and the studied topics in this thesis.

well. For example, new packages are continuously being introduced (Abdalkareem, Oda, et al., 2020; den Besten, Amrit, Capiluppi, & Robles, 2020; Kula, German, Ouni, Ishio, & Inoue, 2017; Wittern et al., 2016), making other packages obsolete, dormant, or even deprecated (Valiev, Vasilescu, & Herbsleb, 2018). Consequently, it is becoming increasingly important for software developers to ensure that they are using well-maintained packages from the ecosystem. Also, as the software evolves (and its dependencies do as well), updating these dependencies can become more challenging (Bogart, Kästner, & Herbsleb, 2015; Decan, Mens, & Claes, 2017; Decan, Mens, & Grosjean, 2019).

Dependency management is essential for maintaining well-functioning projects. As described in Figure 1.1, the dependency management activities typically start by the developers selecting a set of suitable packages from a wide range of available packages in the software ecosystem to perform the required tasks. Then, developers must integrate the packages with their codebase and interact with them through their APIs. Next, developers check for available version updates that could fix bugs, patch security vulnerabilities, introduce new features, or improve the performance of the packages. Finally, to maintain well-functioning dependencies that support software evolution, developers evaluate their dependencies and take required actions to adopt better solutions when necessary (He et al., 2021).

The grey boxes in Figure 1.1 represent the dependency maintenance challenges with proposed mitigations that are the focus of this thesis:

- (1) **How to select packages?** When developers want to use a package to perform a specific task, it is common to find several packages that meet their requirements. Developers must pick only one package. The decision becomes even more challenging when the developers have no experience with using the alternative packages. An empirical study to help developers select packages is presented in Chapter 4.
- (2) **Should we adopt a newer package version?** Packages release new versions to fix bugs, add new features, or patch security vulnerabilities. However, adopting a newly released version risk introducing a breakage behavior. An approach to help developers detect breakage-inducing versions is presented in Chapter 5.
- (3) **Should we migrate from this package?** Packages over time could become vulnerable, obsolete, dormant, or even deprecated (Coelho & Valente, 2017; Valiev et al., 2018). Identifying obsolete packages that should be replaced is not a trivial task, especially when having a long list of dependencies. An approach to identify such packages is presented in Chapter 6.
- (4) **What package should we migrate to?** When a developer needs to replace an obsolete package that has lost community interest, finding the best alternative candidates is essential. An approach to help developers in finding alternatives for packages should be replaced is presented in Chapter 7.

## 1.1 Research Statement

Motivated by the challenges described in the previous section, the goal of this thesis is to propose mitigations for each of the challenges. We believe that crowd wisdom can be

employed to help developers manage their open source software dependencies. We state our research statement as follows:

*Given the challenges of depending on third-party open source packages, developers need techniques to effectively select dependencies, update dependency versions, and avoid outdated dependencies. We employ information extracted from software ecosystems to help developers better manage their software dependencies.*

## 1.2 Thesis Overview

In this section, we provide a brief overview of the thesis. The thesis consists of seven chapters, which can be classified into three main parts. In the first part (Chapters 2 and 3), we provide a background and present the related work. In the second part (Chapter 4), we present an empirical study about the factors that developers should take into consideration when selecting a package. The last part (Chapters 5 to 7), presents our contribution to assist developer in managing their package dependencies. Finally, Chapter 8 concludes the thesis and discusses future work.

### **Chapter 4: An Empirical Study on the Characteristics of Highly-Selected Packages**

With the popularity of software ecosystems, the number of open source packages has been growing rapidly. Identifying high-quality and well-maintained packages from a large pool of packages is an important problem since it is beneficial for various applications, such as package recommendation systems, package search engines, etc. However, there is no systematic and comprehensive work so far that focuses on addressing this problem except in online discussions or informal literature and interviews. Therefore, we investigate the characteristics of highly-used packages. More specifically, in this chapter, we use

a mixed-method approach to examine the characteristic of highly-used npm packages. We start by identifying packages selection factors from the literature review and existing online package search tools. Then, we try to qualitatively understand the factors that developers look for when choosing an npm package to use through surveying JavaScript developers. Next, we quantitatively examine these factors by building a logistic regression model using a dataset of highly-used and low-used packages. Our results indicate that developers mainly consider packages that are well-documented, popular, and do not suffer from security vulnerabilities.

## Chapter 5: An Approach to Identify Breaking Updates

After a developer select and use a package, as any actively maintained project, the package will typically have new versions. The question of whether the developers should update to a newly released version is a vital development decision. On the one hand, updating the package version means that developers will receive the newest features, bug fixes, and security patches (Cadariu, Bouwers, Visser, & van Deursen, 2015; Decan, Mens, & Constantinou, 2018b). On the other hand, the fear of breaking an existing functionality often lingers on the minds of developers (Decan, Mens, & Constantinou, 2018a; Kula et al., 2017; Zerouali, Constantinou, Mens, Robles, & González-Barahona, 2018).

To ensure the stability and quality of newly released dependency versions, developers often run their own tests. This has proven to be a good solution and some tools (e.g., Dependabot<sup>2</sup>, Snyk<sup>3</sup>, Renovate<sup>4</sup> and Greenkeeper<sup>5</sup>) support the automation of such approaches. However, in many cases, developers are still forced to "roll back" dependency updates because they introduce regressions in their system functionality (Cogo, Oliva, & Hassan, 2019). Indeed, Mirhosseini and Parnin (2017) found that there is a need for new

---

<sup>2</sup><https://dependabot.com>

<sup>3</sup><https://snyk.io>

<sup>4</sup><https://renovatebot.com>

<sup>5</sup><https://greenkeeper.io>



techniques to increase the confidence in automated dependency updates. Therefore, in this chapter, we propose a technique to detect breakage-inducing versions of third-party dependencies. More specifically, we propose a crowd-based approach that leverages the automated test suites of other projects that depend on the same dependency to test newly released versions.

Our empirical evaluation shows that the proposed approach can improve the test coverage of the dependencies. More importantly, our approach was able to detect six out of ten real-world breakage-inducing versions that were not detectable by running the test suites from the packages themselves. This work was published in the proceedings of the 17th International Conference on Mining Software Repositories (MSR) [2020](#).

## **Chapter 6: An Approach to Identify Packages in Decline**

Even with updating the dependencies to the latest releases and avoiding breakage-inducing versions as explained in Chapter 5, dependencies could become obsolete ([Valiev et al., 2018](#)). Ecosystems evolve rapidly and developers add new packages every day to solve new problems or provide alternative solutions, causing obsolete packages to decline in their importance to the community. Developers should avoid depending on packages in decline, as these packages are reused less over time and may become less frequently maintained. However, current popularity metrics are not sufficient to provide this information to developers. Thus, the goal of this chapter is to detect packages in decline. More specifically, we propose a scalable approach that uses the centrality of the packages in the ecosystem to identify packages in decline.

Our empirical evaluation shows that our approach aptly detects packages in decline months before they become deprecated or their ranking significantly decreases. Notably, none of the other metrics (i.e., number of dependents, downloads, stars, and forks) provided the same indications. A manuscript based on this work has been accepted in the

Special Issue on Collaboration and Innovation Dynamics in Software Ecosystems, IEEE Transactions on Engineering Management journal (TEM).

## **Chapter 7: An Approach to Find Alternative Packages**

After identifying packages in decline as in Chapter 6, selecting better alternatives is a challenging decision. Knowledge about better solutions requires close involvement or active investigation and searching. Since popular packages in software ecosystems are used by a large base of developers, this chapter aims to use crowd knowledge to suggest better alternative packages. More specifically, we propose an approach that uses the package dependency migration patterns combined with the centrality trends to suggest alternative packages. Then, we empirically evaluate our approach on the npm ecosystem. The evaluation that our approach that the proposed approach accurately suggests alternatives for packages in decline. Also, the evaluation shows that developers support having a tool that utilizes our approach to suggest alternative packages, and the majority of them will use the suggested alternatives.

## 1.3 Thesis Contributions

The main contributions of the thesis are:

- We identify the essential factors that software developers should consider when selecting an npm package to use.
- We propose an approach to detect breakage-inducing versions, which helps developers make more informed decisions before updating to a newly released version.
- We propose an approach that detects packages in decline before they become obsolete or deprecated. Also, we implement a tool that utilizes our approach to make it more accessible for practitioners.
- We propose an approach to find alternative packages to replace packages in decline.

## 1.4 Thesis Organization

This thesis is organized as follows: Chapter 2 provides a background related to this thesis. Chapter 3 presents the literature review. In Chapter 4, we present our empirical study related to selecting packages from software ecosystems. Chapters 5 to 7 propose approaches to support maintaining package dependencies in software ecosystems. Chapter 8 summarizes the thesis and discusses some directions of future work.

# Chapter 2

## Background

In this chapter, we present the concept of the software ecosystem, its related terminologies, and the most popular ecosystems. Also, we explain the package dependencies in the context of the software ecosystems.

### 2.1 Software Ecosystems

A *software ecosystem* is a collection of software packages that are developed, distributed, and evolved in the same environment, e.g., with the same package manager (Lungu, Lanza, Gîrba, & Robbes, 2010). A *package manager* is a software tool that automates the process of installing, configuring, upgrading or removing software packages in a consistent process (Burrows, 2017; Decan et al., 2019). The following are some of the most popular software ecosystems and their package managers:

- CRAN (cran.r-project.org), the Comprehensive R Archive Network. It constitutes the official repository of the statistical computing environment R. The repository hosts more than 18 thousand packages (DeBill, 2021).
- npm (npmjs.com), started in 2010, is the official package registry for the JavaScript

runtime environment Node.js. The npm registry is considered the largest, hosting more than 1.7 million packages (DeBill, 2021).

- PyPI (pypi.org), the Python Package Index, an official third-party software repository for Python. It is the default source of packages for Python's standard package manager (pip). The repository hosts more than 330 thousand packages (DeBill, 2021).

## 2.2 Package Dependencies

Packages in software ecosystems can depend on other packages by reusing some or all of their functionalities. Dependencies can be *direct* (e.g., when a package explicitly depends on another package) or *transitive* (e.g., when a package depends on another package that itself depends on a third package).

### 2.2.1 Semantic Versioning

Semantic Versioning (referred as `semver`) is a simple set of rules and requirements that dictate how version numbers are assigned and incremented for releases of packages. Given a version number `MAJOR.MINOR.PATCH`, it should increment as follows (Preston-Werner, 2020):

- MAJOR version when introduce incompatible API changes.
- MINOR version when add functionality in a backward-compatible manner.
- PATCH version when make backward compatible bug fixes.
- Additional labels for pre-release and build metadata can be added as extensions to the `MAJOR.MINOR.PATCH` format.

Semantic Versioning allows dependent software to be aware of possible “breaking changes” (Bogart, Kästner, Herbsleb, & Thung, 2016). Developers can use *semantic versioning constraints* to allow package managers to automatically apply version updates by specifying the allowed version ranges. For example, a developer can restrict to allow only patch versions and ignore minor and major versions.

Unfortunately, while it is easy to adopt a semantic versioning policy, many studies show that the semantic versioning is not always correctly followed by package developers (Mezzetti, Møller, & Torp, 2018; Møller & Torp, 2019; Raemaekers, van Deursen, & Visser, 2014). In this thesis, we will shed light on the issues related to semantic versioning policy violation and the possible mitigation.

### **2.2.2 Technical Lag**

Managing healthy dependencies require keeping them up-to-date. Hence, ignoring a dependency update creates a technical lag. Technical lag refers to the increasing lag between the latest release of a package and the used version of the same package if no version updating actions are taken. Such technical lag makes the package dependencies outdated, which can be transferred to their dependents as a ripple effect. Developers must balance between the technical lag that their software acquire as time passes and the effort and issues caused by upgrading their dependencies (Gonzalez-Barahona, Sherwood, Robles, & Izquierdo, 2017)

### **2.2.3 Dependency Migration**

The term migration is commonly used in software engineering research. However, the term migration may refer to different development activities. Common cases of migrations in software development include migrating between:

- Programming languages (Bui, Yu, & Jiang, 2019; Dorninger, Moser, & Pichler, 2017;

Nguyen, Tu, & Nguyen, 2016).

- Platforms (Fleurey, Breton, Baudry, Nicolas, & Jézéquel, 2007; Verhaeghe et al., 2019).
- APIs (C. Chen, Xing, Liu, & Xiong, 2021; Gokhale, Ganapathy, & Padmanaban, 2013; Kapur, Cossette, & Walker, 2010; Nita & Notkin, 2010).
- Versions (Cossette & Walker, 2012; Kula et al., 2017).
- Packages (Alrubaye, Mkaouer, & Ouni, 2019a; Teyton, Falleri, & Blanc, 2012; Teyton, Falleri, Palyart, & Blanc, 2014).

In this thesis, we use the term *dependency migration* to refer to the process of replacing one package with another package of similar functionalities, as in many of the previous research (Alrubaye, Alshoaibi, Alomar, Mkaouer, & Ouni, 2020; Alrubaye, Mkaouer, & Ouni, 2019b; He et al., 2021; Kabinna, Bezemer, Shang, & Hassan, 2016).

# Chapter 3

## Related Work

In this section, we present the work most related to this thesis. We divide the prior work into three main areas; work related to the study of reusing packages in software ecosystems, API breakage changes, and API testing.

### 3.1 Reusing Packages from Software Ecosystems

Researchers studied the challenges in managing software ecosystems ([Barbosa, dos Santos, Alves, Werner, & Jansen, 2013](#); [Bosch, 2010](#); [van den Berk, Jansen, & Luinenburg, 2010](#); [Yu & Deng, 2011](#)). Further, the increasing trend of depending on software ecosystems has motivated researchers to understand the developer perspective about using third-party packages.

[Haefliger, von Krogh, and Spaeth \(2008\)](#) studied the reuse pattern and practices in open source applications. Their study shows that experienced developers reuse more code than less experienced developers. [Abdalkareem et al. \(2017\)](#) studied an emerging code reuse practice in the form of lightweight packages in the software ecosystem. Their study was conducted to understand why developers use trivial packages. Their results showed that



these packages are prevalent in PyPI (Python Package Index), but 70.3% of the developers consider using these packages is a bad practice. [Xu, An, Thung, Khomh, and Lo \(2019\)](#) studied the reason behind the reusing and re-implementing of external packages in software applications. They found that developers often replace their self-implemented methods with external libraries because they were initially unaware of the library or it was unavailable back then. Later on, when they become aware of a well-maintained and tested package, they replace their own code with that package. Although developers prefer to reuse code than re-implement it, they replace an external heavy package with their implementation when they believe that they are only using a small part of its functionalities or if it becomes deprecated. [Haenni, Lungu, Schwarz, and Nierstrasz \(2013\)](#) conducted a survey with developers about their decision-making while introducing a dependency to their applications. Surprisingly, the study found that developers generally do not apply rationale while selecting the packages; they use any package that accomplishes the required tasks. More recently, [Y. Ma, Mockus, Zaretski, Bichescu, and Bradley \(2020\)](#) performed an empirical study to investigate how developers choose between two comparable packages.

Other work also focused on examining the popularity growth of packages within an ecosystem. For example, [S. Qiu, Kula, and Inoue \(2018\)](#) studied the growth of popular npm package. Their finding shows that lifetime, number of dependents, and added new functionalities play significant roles in popularity growth. [Chatzidimitriou., Papamichail., Diamantopoulos., Oikonomou., and Symeonidis. \(2019\)](#) use network analysis and information retrieval techniques to study the dependencies that co-occur in the development of npm packages. Then, they use the constructed network to identify the communities that have been evolved as the main drivers for npm's exponential growth. Their findings show that several clusters of packages can be identified. [Zerouali, Mens, Robles, and Gonzalez-Barahona \(2019\)](#) examined a large number of npm packages by extracting nine popularity metrics. They focused on understanding the relationship between the popularity metrics.

They found that the studied popularity metrics were not strongly correlated. [Dey, Kar-nauch, and Mockus \(2021\)](#) proposed the concept of skill space which allow measuring distances between APIs, repositories, programming languages, and developers to provide the ability to assess relationships among them.

Other work focused on understanding the process used by developers to select packages and attempted to provide some guidelines. [Pano, Graziotin, and Abrahamsson \(2018\)](#) focused on understanding factors that developers look for when selecting a JavaScript framework (e.g., react). Based on interviewing 18 decision-makers, they observed a list of factors when choosing a new JavaScript framework, including the community's size behind the framework. [del Bianco, Lavazza, Morasca, and Taibi \(2011\)](#) provided a list of factors that influence the trustworthiness of open source software components. Their list had five categories, including quality and economic categories. Also, in their study, [Hauge, Osterlie, Sorensen, and Gereaa \(2009\)](#) observed that many organizations apply informal selection process based on previous experience, recommendations from experts, and information available on the Internet. [Franch and Carvallo \(2003\)](#) adapt the ISO quality model and assign metrics to be used as a measure for selecting software components. Their study suggested that relationships between quality entities must describe explicitly.

**The main goal of our work in Chapter 4 is to examine the characteristics of highly-selected packages within the npm ecosystem.** In many ways, our study is complementary to prior work because we focus on understanding factors that make a package highly-selected. Our study is one of the only studies to use mixed research methods, which provide us with a more complete and synergistic utilization of data than any separate quantitative and qualitative data collection and analysis.

## 3.2 Studying API Breakage Changes

Several studies investigated API evolution and stability and proposed techniques to detect breakage changes (Dig & Johnson, 2006; Kapur et al., 2010; Mostafa, Rodriguez, & Wang, 2017; Xavier, Brito, Hora, & Valente, 2017). Recently, Xavier et al. (2017) performed a large-scale analysis on 317 real-world Java libraries with 9K releases and 260K client projects. Their results show that 14.78% of the API changes are incompatible with previous versions and 2.54% of their clients are impacted. They also found that libraries with a higher frequency of breaking changes are larger, more popular, and more active. Bogart et al. (2016) empirically studied three software ecosystems, including npm, and found that fixing bugs, efficiency improvements, and addressing technical debt are the main reasons for inducing breakage changes API. Also, Businge et al. (Businge, Serebrenik, & van den Brand, 2012, 2015) studied Eclipse interface usage by Eclipse third-party plug-ins and evaluated the effect of API changes and non-API changes. Dig and Johnson (2006) proposed a catalog of API breaking changes and non-breaking changes. They found that 80% of the changes that break dependent projects are related to refactoring tasks.

Raemaekers et al. (2014) investigate the use of semantic versioning in Java libraries. They found that breakage changes are prevalent in Java libraries. Zhong and Mei (2018) conducted an empirical study on API usages focusing on how different types of APIs are used. Their empirical results showed that single API class usages are mostly strict orders, while multiple API class usages are more complicated because they include both strict orders and partial orders. Also, Kula et al. (2017) studied more than 4,600 open source projects and found that 81.5% of studied projects are keeping their outdated dependencies libraries.

Mostafa et al. (2017) performed an investigation to gain insight on the behavioral backward incompatibilities of Java libraries. They proposed a method that use regression testing of 68 version pairs of 15 Java libraries, and examine more than 120 real world bugs.

Their results showed that behavioral backward incompatibilities are not well understood by libraries developers and rarely documented. [Kim, Nam, Yeon, Choi, and Kim \(2015\)](#) propose a tool called `Remi` that predicts high-risk APIs in terms of producing potential bugs in the dependent projects. The main goal of `Remi` is to assist developers in writing more test cases for the high risky APIs. [Rodríguez-Baquero and Linares-Vásquez \(2018\)](#) present the use of 43 mutation test operations to test Node.js and JavaScript projects and leverage the npm platform to run test suites. They found that the proposed operations provide a mutation test coverage of 70.59% on average. [Taneja, Zhang, and Xie \(2010\)](#) proposed an automated test generation for database applications using mock objects, demonstrating that with this technique they could achieve better test coverage. [Abdalkareem et al. \(2017\)](#) studied the use of trivial packages on npm and found that even though developers believe that trivial packages on npm are well-test, their qualitative analysis showed that only 45% of the trivial packages have test case written for them.

**The work by [Mezzetti et al. \(2018\)](#) is closest to our work in Chapter 5.** In their work, the authors proposed a technique to detect packages that break the types of their public interface in the npm ecosystem. The study leverage the test suites of dependent projects and uses a dynamic analysis to learn models of the package interface types. Our work complements the prior work because we propose a technique that leverages tests from dependent projects to detect semantic and behavioral breakage-inducing versions of target dependency.

### 3.3 Studying Package Evolution in Software Ecosystems

Several studies examine the overall growth of software ecosystems. For example, [Wittem et al. \(2016\)](#) did the first large-scale study of the npm ecosystem. They study the evolution of the npm ecosystem regarding growth and development activities. The study found that only 27.5% of packages in the npm ecosystem are depended upon, indicating

that developers largely depend on a core set of packages. [Decan et al. \(2019\)](#) also empirically compare the evolution of the dependency network in seven software packaging ecosystems. Their results show how fast each packaging ecosystem and packaging dependency network is growing over time. They observe the continuing growth of the number of packages and their dependency relationships. Some other work also studies the evolution of software ecosystems (e.g., ([German, Adams, & Hassan, 2013](#); [Kikas, Gousios, Dumas, & Pfahl, 2017](#))). In the same line with these existing studies, our work in Chapter 6 examines the evolution of npm ecosystem in terms of its dependency graph. However, we focus on employing the npm dependency graph and calculate the centrality for each package to identify npm packages that are in decline.

Other work has been done to examine software projects that are not active anymore. For example, [Coelho, Valente, Silva, and Shihab \(2018\)](#) use machine learning classifiers to identify unmaintained GitHub projects. They also examine the level of maintenance activity of active GitHub projects to detect unmaintained projects. In a following work, [Coelho, Valente, Milen, and Silva \(2020\)](#) developed a metric to alert developers about the risks of depending on a given GitHub project based on the built ML classifiers. In the context of the Python ecosystem, [Valiev et al. \(2018\)](#) studied the factors that affect the sustainability open source projects. Their results show that the centrality of a project in the ecosystem dependency network has a high impact on the project activities. Other works also investigate the overall popularity of open source projects. For example, [Borges, Hora, and Valente \(2016\)](#) studied the popularity of GitHub repositories. They identified four patterns of popularity growth, which relate to factors such as stars and forks. As shown in the work mentioned above, examining the level of activity of an open source project is of critical importance, in particular, for packages in software ecosystems to maintain healthy dependencies. **Hence, our work in Chapter 6 addresses this issue by detecting which npm packages are in decline.**

There is also a body of research that investigates specific aspects of packages in a software ecosystem, including the source code size of packages (Abdalkareem, Oda, et al., 2020), the impact of forks on the popularity of packages (Zhu, Zhou, & Mockus, 2014), conflicts between used JavaScript packages (Patra, Dixit, & Pradel, 2018) or Python packages (Y. Wang et al., 2020), identifying breaking updates in npm package (Mezzetti et al., 2018; Møller & Torp, 2019), and studying cross-project bugs that may impact a large part of a software ecosystem (W. Ma et al., 2020). Similar to these aforementioned studies, we focus on one aspect of the used packages in the npm ecosystem: the package centrality. We propose in Chapter 6 the use of package centrality to identify packages in decline and evaluate its effectiveness in the npm ecosystem.

### 3.4 Recommending Package Dependencies

Several studies proposed approaches to recommend packages to developers. Thung, Lo, and Lawall (2013) proposed an approach to recommend packages for projects based on their current dependencies, using association rule mining and collaborative filtering. Other studies targeted the same problem by using different approaches, such as multi-objective optimization (Ouni et al., 2017), and pattern mining and hierarchical clustering (Saied et al., 2018). Recently, Nguyen, Di Rocco, Di Ruscio, and Di Penta (2020) proposed a more efficient approach as it generates recommendations in a comparably less historical data. The main goal of these approaches is to tap in the missed opportunities of using available packages, based on the project's package dependencies and characteristics. **However, our goal in Chapter 7 is to recommend migration opportunities of better alternatives than packages already in use.**

Chen, Gao, and Xing (2016) proposed an approach for mining Stack Overflow tags to find semantically similar packages. Even though this approach can return a set of similar packages, it has no evidence of the feasibility of migrations between the alternatives (He et

al., 2021). Thus, researchers proposed mining historical migrations from existing software repositories, which rely on the crowd's wisdom in performing migrations to alternative packages (Alrubaye et al., 2019a; Teyton et al., 2012, 2014). However, their approaches suffer from either low recall (Alrubaye et al., 2019a; Teyton et al., 2012), or low precision (Teyton et al., 2012, 2014). He et al. (2021) improved the performance by utilizing multiple metrics to capture different dimensions of evidence from development histories when recommending dependency migrations extracted from other software repositories. This approach relies on analyzing the commits and their messages to extract migration patterns, it is sensitive to how developers divide their changes across commits and the clarity of commit messages. In contrast, our approach extracts dependency migrations based on versions without considering the individual commits, which overcome the previous limitation. **Also, our approach in Chapter 7 targets migration suggestions for packages in decline only, avoiding overwhelming the developers with undesired suggestions (Erlenhov, Neto, & Leitner, 2020).**

Researchers empirically investigated dependency migrations. Kabinna et al. (2016) highlight the challenges in migrating to new logging packages. Alrubaye et al. (2020) analyzed several code quality metrics before and after applying dependency migrations. de la Mora and Nadi (2018) studied the role of common metrics in developer selection of packages. He et al. (2021) proposed new four metrics (i.e., Rule Support, Message Support, Distance Support, and API Support) to rank the migration suggestions. They all use project level metrics in their approaches, while we are the first to use the centrality in the context of dependency migrations (i.e., an ecosystem level metric), which emphasizes the community interest in performing the dependency migrations.

# Chapter 4

## An Empirical Study on the Characteristics of Highly-Selected Packages

### 4.1 Introduction

In recent years, the proliferation of software ecosystems has led to a vast and rapid growth of the number of open source packages.<sup>1</sup> As of September 2021, there were over 1.7 million packages available on the registry of the Node Package Manager (npm), one of the largest software ecosystems. Furthermore, the number of packages grew by around 60% between September 2019 and September 2021 ([DeBill, 2021](#)).

With the massive number of packages out there, finding the right package to use can be challenging, considering that many packages provide similar functionalities. However, there are packages that stand out and experience high interest from developers. We believe that understanding the characteristics of these highly-selected packages is very important

---

<sup>1</sup>In this chapter, we use the term package referring to open source components published on software ecosystems.



since it helps developers answer the question: which packages a developer should choose among many existing choices. In addition, it can be used to improve the performance of package recommendation systems (de la Mora & Nadi, 2018; Semeteys, 2008; StackOverflow, 2017; Zheng, Zhang, & Lyu, 2011) and enhance the user experience of package search engines (Abdellatif, Zeng, Elshafei, Shihab, & Shang, 2020; Cruz & Duarte, 2018; StackOverflow, 2017). For package developers, understanding the factors of choosing highly-selected packages can be helpful for various purposes, such as improving the aspects that the developer look for, acquiring the community attention, and eventually increase the package usage. The potential implications of understanding these factors motivate our work.

Previous studies examine different aspects of packages in software ecosystems, such as centrality and popularity (Abdalkareem, Oda, et al., 2020; Abdellatif et al., 2020; Larios Vargas, Aniche, Treude, Bruntink, & Gousios, 2020; S. Qiu et al., 2018), and some examine the selection factors of relevant packages (Jadhav & Sonar, 2009; Larios Vargas et al., 2020). The main limitation of prior works is that they are based on a purely quantitative analysis of popular packages or only interviewing developers in a specific industrial context. In addition, understanding the characteristics of highly-selected packages is still the subject of much discussion and refinement. This is because several facts include personality aspects and examining different data modalities from several sources, in which a developer is typically a familiar user of a specific package. Thus, in this chapter, we divide the study into two parts - qualitative and quantitative (John, Creswell, & CLARK, 2000). Figure 4.1 shows an overview of our study design. In the first part (referred from now on as *qualitative analysis*), we conduct a user study survey that involves 118 JavaScript developers. We ask our survey participants to fill-in a form composed of 17 statements about factors they use when selecting npm packages. Then, we qualitatively analyze the answers to the 17 questions using descriptive statistics.

**1. Qualitative Analysis:** surveying Javascript developers about package usage factors.



**Survey:** 118 JavaScript Developer responses.

**2. Quantitative Analysis:** using a logistic regression model to validate the developers' perceptions.



**Github Data:** we collected repository level factors.



**Snyk Data:** we collected security vulnerability factors.



**npm Data:** we collected package level factors.

Figure 4.1: An overview of our study design.

To provide explanations to the findings of the qualitative analysis, we conduct *quantitative analysis* on a set of 2,427 npm packages grouped into highly-selected and not highly-selected packages. Similar to prior work ([Bavota et al., 2015](#); [Lee et al., 2020](#); [Tian, Nagappan, Lo, & Hassan, 2015](#)), we estimate the highly-selected packages based on the number of dependent packages (i.e., clients packages) within the npm ecosystem. Then, we analyze the selected packages and collect quantitative data to present the factors studied in our survey. Next, we use regression analysis to quantitatively explain which of the studied factors are the most important.

The survey results show that JavaScript developers believe that when selecting a package to use, they look for packages that are: well-documented, receive a high number of stars on GitHub, have a large number of downloads, and do not suffer from security vulnerabilities. Moreover, our regression analysis complements the results of our survey about highly-selected packages. Also, it describes the differences between the developers' perceptions about highly-selected packages and the characteristics of highly-selected packages. In general, our work makes the following key contributions:

- We perform a mixed qualitative and quantitative analysis to investigate the characteristics of highly-selected packages on the npm ecosystem. We present our results from

surveying 118 JavaScript developers and validate the survey result through a quantitative analysis of 2,427 npm packages.

- We identify the most important factors that packages' users should consider when selecting an npm package to use in their projects.
- We provide practical implications for packages' maintainers, the npm ecosystem's maintainers, and researchers and outline future research avenues.

The remainder of this chapter is structured as follows. Section 4.2 describes the study design and presents the results of the qualitative analysis. Section 4.3 describes the study design and presents the results of the quantitative analysis. We discuss the implications of our study in Section 4.4. We discuss the threats that may affect the validity of the results in Section 4.5. Finally, Section 4.6 concludes our work.

## **4.2 Qualitative Analysis**

This analysis aims to survey JavaScript developers to understand the characteristics of packages that JavaScript developers look for when selecting an npm package to use. In this study, we surveyed 118 JavaScript developers.

### **4.2.1 Study Design**

This section presents our survey design, participant recruitment, and data analysis methods.

#### **Survey Design**

To understand which factors developers look for when selecting an npm package, we design a survey containing three main parts. The first part contains questions related to the

background of the participants. We ask these questions to ensure that our survey participants have sufficient experience in software development and in selecting and using npm packages. In this part, we ask the following questions:

- (1) How would you best describe yourself? A question with the following choices and the last choice is a free-text form: Full-time, Part-time, Free-lancer, and Other.
- (2) For how long have you been developing software? A selection question with the following options: <1 year, 1–3, 4–5, more than 5 years.
- (3) How many years of JavaScript development experience do you have? A selection question with the following options: <1 year, 1–3, 4–5, more than 5 years.
- (4) How many years of experience do you have using the Node Package Manager (npm)? A selection question with the following options: <1 year, 1–3, 4–5, more than 5 years.
- (5) How often do you search for npm packages? A question with the following options: Never, Rarely (e.g., once a year), Sometimes (e.g., once a month), Often (e.g., once a week), Very often (e.g., everyday).
- (6) Which search engine interface do you use to find relevant npm packages? A question with the following multiple choices and the last choice is a free-text form: Online search on the npm web page (i.e., *npmjs*), Command line search, Google or other general web search engines, and Other.

In the second part of the survey, we have a list of statements that present seventeen factors that can affect selecting npm packages. In particular, we ask the question “*How important are the following factors when selecting a relevant npm package?*” Table 4.1 reports the seventeen factors statements. For each statement, the table presents each factor’s definition and the rationale behind asking about it. In the survey, we ask participants to rate these statements using a Likert-scale ranges from 1 = not important to 5 = very

Table 4.1: List of factors used in selecting a packages from the npm ecosystem.

Factor	The survey statements	Rationale
Forks	The number of forks for the package's source code on GitHub.	The number of forks that a package receives gives an indication that the packages are active and many developers are contributing to these packages (Gousios, Pinzger, & Deursen, 2014).
Watchers	The number of watchers of the package's GitHub repository.	Developers can watch package repositories on Github so they can receive notifications about package development activities (Sheoran, Blincoe, Kalliamvakou, Damian, & Ell, 2014). The higher the number of watchers on a package indicates that the package is well-known and used by many developers. We consider that packages with an increased number of watchers refer to highly-selected packages.
Contributors	The number of contributors to a package's GitHub repository.	The higher the number of contributors to a package shows that the package is more likely to attract developers (Yamashita, Kamei, McIntosh, Hassan, & Ubayashi, 2016). Thus, it indicates that the package is highly-selected.
Downloads	The number of downloads the package has.	The package that has a higher number of downloads indicates that the packages to selected and used (Abdellatif et al., 2020).
Stars	The number of stars of a packages on GitHub.	The npm package that received a high number of stars on GitHub could indicate to developers that a package is more likely popular, which may attract them to use the package (Borges & Valente, 2018; Dabbish, Stuart, Tsay, & Herbsleb, 2012).
Dependencies	The number of dependencies the npm package has.	A larger number of dependencies could not attract more developers to use the packages since prior work shows that packages with a more considerable dependency may lead to dependency hell (Abdalkareem et al., 2017).
License	Whether the npm package has a permissive or restrictive software license.	When evaluating a package, it is also essential to consider non-functional requirements, such as the license. Using a package with no license or with a license that does not match the developer organization's usage and policies can quickly become a problem (Meloca et al., 2018; Team, 2019).
Documentation	Whether the npm package repository has online documentation, e.g. README file.	The package that is well-documented and has a very organized README file is more likely to be used by many developers (Begel, Bosch, & Storey, 2013; Hata, Todo, Onoue, & Matsumoto, 2015).
Test Code	Whether the npm package has test cases written.	Packages that have test code written are more likely to attract developers to use them since it indicates that the packages are well-tested (Abdalkareem et al., 2017).
Build Status	The build status of the npm package for example from Travis CI.	The presence of a high number of failed builds in the package repository may lead developers not to use the package (Abdellatif et al., 2020).
Vulnerabilities	If the npm package depends on vulnerable dependencies.	If a npm package is affected by vulnerabilities, it may concern developers and deter them from using the package (Abdalkareem, Oda, et al., 2020; Abdellatif et al., 2020).
Badges	If the package repository has badges.	The presence of badges in the package repository indicates that the package is of good quality that attracts developers to use the package (Trockman, Zhou, Kästner, & Vasilescu, 2018).
Website	If the package has a custom website.	The presence of a website for the package indicates that the package is supported by an organization, which is usually a signal that there is more than one maintainers or major contributor (i.e., there is support by an organization) (H. S. Qiu, Li, Padala, Sarma, & Vasilescu, 2019).
Releases	The release frequency of the package.	A package with several releases indicates that the package is well maintained, which may increase the application's maintenance overhead.
Closed Issues	The number of closed issues in the package's repository.	The number of closed issues indicates how well-maintained the package is and reveals how maintainers of the package respond to issues. Packages with a large percentage of closed issues attract more developers to use the package (Abdellatif et al., 2020).
Commit Frequency	The commit frequency in the package repository.	Developers mainly look for well-maintained and active packages to use. Prior work also shows that the number of commits a package receives gives a good indication of how active the package is, which results in high usage (Abdellatif et al., 2020).
Usage	The number of projects using the package on GitHub.	Packages that are used by many other developers are more likely to attract more developers to use (Abdalkareem, Oda, et al., 2020).

important ([Oppenheim, 1992](#)). We choose to investigate these factors since: 1) our literature review indicates that these factors are known to impact the usage and selection of npm packages, 2) we focus on studying factors that developers can easily observe through examining the package source code or its software repository, e.g., from the GitHub website.

In the last part of our survey, we ask the participants an open-ended question about whether they have any additional comments or other factors that they look for when they select a package. We ask this open-ended question to give our survey participants maximum flexibility to express their opinion and experience with the selection of npm packages, which also comply with the survey design guidelines ([Dillman, 2011](#)).

Once we had our survey questions, we shared the survey with three colleagues who are experts in JavaScript developers who use packages from npm. We did this to discover potential misunderstandings or unexpected questions early on and improve our survey ([Dillman, 2011](#)).

## **Participant Recruitment**

To identify the participants in our survey, we need to reach out to developers who are the experts in selecting and using JavaScript packages. Thus, we resort to the public registry of npm ([npm, 2017b](#)). The registry contains information on each package published on npm, including the developers maintaining the package. We use the npm registry to collect a list of emails and names of JavaScript developers who use a large number of npm packages. To do so, we analyze the npm registry, and for each package, we extract its dependencies and the contacts of the developers who maintaining the package. Then, we select the top thousand developers based on the number of their distinct package dependencies. It is important to note that we select developers who use a high number of packages since they likely went through the process of selecting npm packages many times.

Once we identified this initial sample of developers, we examined all the names and

Table 4.2: Participants’ position, experience in software development, JavaScript and usage of the npm package manager.

Developers’ Position	Occurrences	Development Experience	Occurrences	Experience in JavaScript	Occurrences	Experience in Using npm	Occurrences
Full-time	84	< 1	2	< 1	0	< 1	0
Part-time	9	1 - 3	21	1 - 3	25	1 - 3	59
Freelancer	15	4 - 5	15	4 - 5	37	4 - 5	20
Other	10	> 5	80	> 5	56	> 5	39

email addresses of the identified developers to exclude duplicated emails and names. Based on this step, we identified 931 unique JavaScript developers. Next, we sent email invitations of our survey to the 931 unique developers. However, since some of the emails were returned for several reasons (e.g., invalid emails), we successfully reached 895 developers. In the end, we received 118 responses for our survey after having the survey available online for ten days. This number of responses translates to a 13.18% response rate, which is comparable to the response rate reported in other software engineering surveys (Smith, Loftin, Murphy-Hill, Bird, & Zimmermann, 2013).

### Survey Participants

Table 4.2 shows the positions of the participants, the development experience of the participants, the JavaScript experience of the participants, and their experiences in using npm ecosystem.

As for the participants’ positions, 84 participants identify themselves as full-time developers and 9 participants as part-time developers. Interestingly, 15 participants identify themselves as freelancers. The remaining ten participants identify themselves as having other positions not listed in the question including, open source developer, IT specialist, and PhD student.

Of the 118 participants in our survey, 80 participants have more than 5 years of development experience and 15 responses have between 4 to 5 years. Also, 21 participants claim to have between 1 to 3 years of experience, and only two participants have less than two

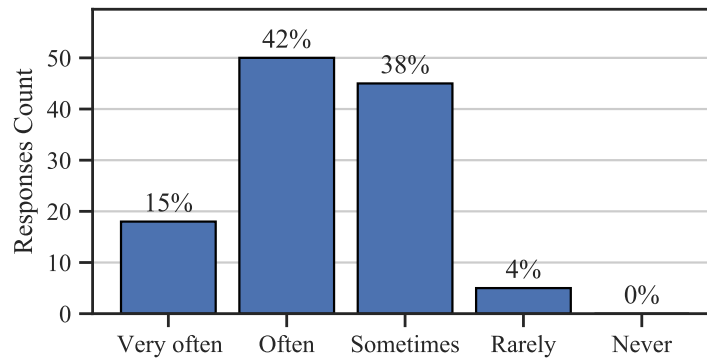


Figure 4.2: Survey responses regarding how often our survey participants search for npm packages. In our survey, the question has the following answers: never, rarely (e.g., once a year), sometimes (e.g., once a month), often (e.g., once a week), very often (e.g., everyday).

years of development experience. In addition, 56 participants have more than 5 years of experience in JavaScript, 37 participants have experience in using JavaScript between 4 to 5 years, and 25 participants claim to have between 1 to 3 years of experience.

We also ask our survey participants about their experience in using packages from the npm ecosystem. The majority of our survey participants indicate that they have more than one year of experience using npm. Specifically, 39 participants have more than 5 years of experience using npm and 20 responses have between 4 to 5 years. Also, 59 participants claim to have between 1 to 3 years of experience.

In addition, to inquire our survey participants about their development experience, we ask them how often they search for npm packages and which search engine they used to perform their search. Figure 4.2 reports the result related to participants's habits about how often they search for npm packages. Of the 118 participants, 15% indicate that they search for npm packages very often, and 42% indicate that they often search for new npm packages. Almost all other participants (38%) indicate they sometimes look for npm packages. Interestingly, only 4% of our survey participants report that they rarely do search for packages, and no one indicates they she/he never looks for npm packages. Our survey participants also report that they use mainly web search engines (e.g., Google) when they



search for npm packages to use. Interestingly, only 20% of them indicate they use other search engines.

Overall, the background information about the developers who participated in our survey shows that they are experienced in JavaScript and selecting npm packages, which gives us confidence in the finding based on their experiences.

## **Analysis Method**

To analyze our survey responses about the different factors used to select npm packages, we show the distribution of the Likert-scale for each factor, which ranges from 1 = not important to 5 = very important (Oppenheim, 1992). Also, for responses of each factor, we calculate values of the median, the interquartile range (IQR), the mean, and the standard deviation (SD).

In addition, to analyze the free-text answers from the open-ended question related to developers' opinions, we perform an iterative coding process to understand whether the responses show any other factors that we did not consider in our survey (Rea & Parker, 2014). The first two authors iteratively developed a set of codes based on an inductive analysis approach (Seaman, 1999). In total, the authors manually examined 30 responses from the developers who answered the optional open-ended question. Based on this analysis, we did not find any new factors that we did not consider in our survey. In fact, all the responses to this open-ended question support the developers' opinions about the studied factors.

### **4.2.2 Study Results**

Table 4.3 shows the factors' name and their survey statements and 5-point Likert-scale distribution for each factor from our survey responses. The table also shows the scale's median alongside IQR and mean alongside SD. Overall, based on the results, we can divide the

Table 4.3: Survey results of the factors used in selecting a package from the npm ecosystem.

Factor	Distribution	Median	IQR	Mean	SD
	1 2 3 4 5				
Documentation		5.0	0.0	4.64	0.77
Downloads		4.5	1.0	4.30	0.88
Stars		4.0	2.0	3.97	1.13
Vulnerabilities		4.0	2.0	3.70	1.24
Release		4.0	1.0	3.48	1.10
Commit frequency		3.0	1.0	3.33	1.23
Closed issue		3.0	1.0	3.29	1.09
License		3.0	3.0	3.26	1.39
Usage		3.0	2.0	3.19	1.33
Test Code		3.0	2.0	3.14	1.31
Dependencies		3.0	2.0	3.12	1.33
Contributors		3.0	2.0	3.03	1.40
Build Status		3.0	2.0	2.89	1.31
Website		3.0	3.0	2.67	1.32
Watchers		3.0	3.0	2.53	1.25
Badges		2.0	2.0	2.25	1.20
Forks		2.0	2.0	2.12	1.25

factors used by developers when selecting packages into three groups: 1) important factors (e.g., documentation, downloads, and stars), 2) somewhat important factors (e.g., license and testing), and 3) unimportant factors (e.g., watchers and badges). In the following, we discuss the developers’ perceptions in more detail:

**Documentations:** on a 5-point scale, participants indicated that the most important factor when looking for an npm packages to use is how well a package is documented. Table 4.3 shows that the majority 93% (median = 5.00 and mean = 4.65) of the responses agree with the statement that the GitHub repository of a npm package that they are examining should have some online documentation. In addition, to confirm this statement, developer P40 stated that “*Sample code documentation on its usage*” are important factors when selecting an npm package to use. Interestingly, one of the developers (P44) mentioned that the quality is an essential factor when they examine the documentation of a

package: *“the documentation quality, many sites have generic readme that does not help.”*

**Downloads:** the second most important factor reported by our survey participants is the number of downloads that the packages have. More than 85% (median = 4.5 and mean = 4.30 on the 5-point scale) of the responses say that they consider the number of downloads a package has when searching for a package to use. These results give a high indication that developers still consider the download count of packages as a sign of the community interest, which means that the package is a good option to select.

**Stars:** our survey showed that developers also look for the number of stars that the packages have. On the 5-points scale, developers believe that the reputation of the packages in terms of start count is an important indicator with median = 4 and mean = 3.97. For example, developer P74 states that *“reputation/popularity”* are the most important factors when selecting an npm package to use.

**Vulnerabilities:** the fourth most important factor developers consider when searching for a new npm package to use is that the packages do not depend on vulnerable or outdated dependencies. On a 5-point scale, 62% of the developers see vulnerabilities as an essential factor when finding relevant npm packages. Furthermore, some developers emphasize the essentiality of this factor, participant P58 said *“... not dependent on other out of date or vulnerable packages.”* Also, participant P45 stated that they look for packages that are free of vulnerable code and use tools to scan for vulnerabilities such as Snyk and dependabot tool.

Our survey also reveals that there are some other factors that developers do not have an agreement on whether they are essential when they search for packages to use or not. We found that factors such as release (median = 4.0 and mean = 3.48), commits frequency (median = 3.0 and mean = 3.33), and test code (median = 3.0 and mean = 3.14) do not have a consistent agreement between the participants in our survey. However, some participants explicitly highlighted the importance of some of these factors, such as developer

P69, who said “*examining the package repository and see the recent and historical activity/commits/updates would help making the decision*”. Another developer, P1 explain that “. . . *the test coverage status is useful, but can be verified manually in the code when deciding to use the package. The last date a commit was made is very important. The more recent the better. The last date a release was made is very important. The more recent the better.*” In addition, we observe from Table 4.3 that developers do not have a consistent agreement about factors such as license and number of dependent applications, number of dependencies that the package uses, the number of closed issues, and the number of contributors, which have, on a 5-point scale, values with a mean of 3.26, 3.19, 3.12, and 3.29, respectively.

The other interesting group of the studied factors that developers tend not to consider when examining an npm package to use are: forks, badges, watchers, website, and build status. Our analysis shows that these factors received median values between 3.0 and 2.0 and mean values between 2.89 and 2.12 on 5-point scale. However, only one developer from our survey supported the idea that examining the build status is essential when selecting a package to use and said P1 “*The build status is important no matter if it comes from Travis CI or other providers ...*”.

Finally, we found that developers use some other factors when looking for npm packages. Our survey participants indicated that if there is a big software company that supports the package. For example, developers P39 said “*The source of the package, if it is by a company that actively supports open source and maintains their open source packages (ex: Facebook, Formidable labs, Infinite Red), brings more points*”. Also, another participant stated the same, such P11 “*Private support for big companies in open source projects or libs (angular-google, react-facebook, etc) that means the package usually follow good practices, test, linter, ci, etc, and the team that maintains the package is really good.*”

In addition, two other developers in our survey indicated that support of community

discussions about the packages matters. For example, P94 mentions “*Whether the package is actively maintained by developers well known and reputed in the community & whether the package has good typescript support*” and P60 said “*If the library is supported with an online community where usage is discussed*”. Another developer, P20 stated “*References on other professional webpages about the package*”.

In summary, JavaScript developers have access to a wealth of information about a large number of npm packages that can be used when deciding which packages to use. Our survey shows that developers mainly consider packages that are well-documented, popular, and do not suffer from security vulnerabilities. Moreover, when we conducted our survey, among the 118 respondents, 73 (62%) provided their emails and showed interest in our findings. This indicates the strong relevance and importance of the findings to the practitioners and the overall JavaScript development community.

### **4.3 Quantitative Analysis**

The goal of this analysis is to triangulate our qualitative findings. In particular, we want to quantitatively validate the developers’ perception about the factors that highly-selected npm packages possess. In this analysis, we examined 2,592 npm packages divided into highly-selected and not highly-selected packages. For each package in our dataset, we collected quantitative data to present the factors studied in our survey. Then, we used regression analysis to quantitatively investigate which of the studied factors are the most important.

### 4.3.1 Study Design

In this section, we describe our methodology of collecting a dataset of highly-selected and not highly-selected npm packages. We also describe how we collect the studied factors, which serve as the dependent variables in our study. Finally, we present our analysis method and steps.

#### Data Collection

To quantitatively examine the factors that make some npm packages highly-selected, we want to have a sufficient number of packages that present both highly-selected and not highly-selected packages. To do so, we resort to study packages from the npm ecosystem. We start by retrieving the metadata information of all the npm packages that are published on the npm ecosystem. In particular, we wrote a crawler to interact with the npm registry and download the package.json file of every npm package as of December 23, 2020 (npm, 2017b). It is important to note that the package.json contains all the package information, including the names of other packages that the package depends on them. Once we have the package.json, we start recursively analyzing the package.json file of every package to extract its dependencies. After that, for each package in the npm ecosystem, we count the number of other packages that list it as a dependency, i.e., number of dependent packages.

It is important to note that we choose to use the number of dependent packages as a proxy of highly-selected packages over other measurements, particularly download count, for two main reasons: 1) npm provides an accumulated download count over time, 2) the download count that npm provides could include crawlers and downloads due to transitive dependencies. Furthermore, our process consider only the direct dependent packages from the npm registry, avoiding including dependent applications from other platforms like GitHub. We do this since our goal is to proxy how many times a developer went through

the process of selecting a package and decided to select the subject package. However, platforms like GitHub hosts millions of applications created using predefined project templates or bootstrapping tools. For example, as of October 5th, 2020, the tool *Create React App* alone bootstrapped 4.8 million public applications on GitHub. Considering such applications will amplify the decision taken by the creators of such tools or templates to overtake the decisions of millions of developers. For the same reason, we do not consider the number of transitive dependents because it does not reflect how many times developers have selected a package.

In total, we analyzed the package.json file of 1,423,956 npm packages. After that, we choose to study 6,924 packages that have more than 100 dependent packages, i.e., the number of packages that depend on the selected packages. We decided to study npm packages that have more than 100 dependent packages for two main reasons. First, we found that prior work indicates that npm ecosystem has many packages that are not used, e.g., toy packages. Thus, selecting packages with more than 100 dependent packages eliminates incompetent packages. Second, since we want to examine highly-selected npm packages, we focus on packages that can potentially be used and appear as an option for developers when searching for an npm package to use, for example, packages that are adopted by other packages. In addition, we select this threshold after examining the distribution of number of dependent packages across all packages in the npm ecosystem.

Next, we sorted the selected npm packages based on their number of dependent packages ([Chatzidimitriou. et al., 2019](#)). Figure 4.3 presents the distribution of the number of dependent packages. We consider the top 20% as highly-selected packages and the bottom 20% as not highly-selected packages. We resort to using these thresholds to have an essential distinction between the two samples, which element gray area between them. Also, prior studies used a similar sampling technique ([Bavota et al., 2015](#); [Lee et al., 2020](#); [Tian et al., 2015](#)). In the end, we had 1,385 highly-selected packages and 1,385 not highly-selected

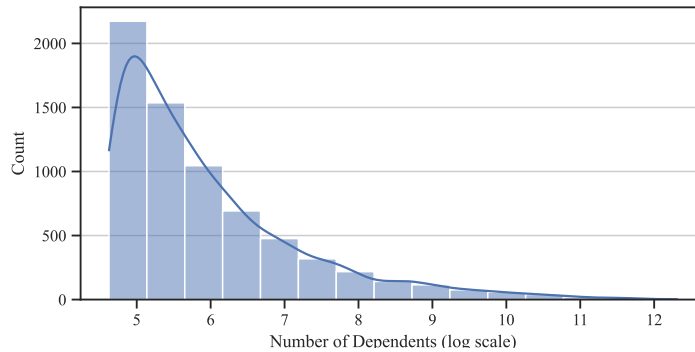


Figure 4.3: A histogram for the used npm packages.

packages. We use these packages in quantitative analysis.

### Package Usage Factors

Since we want to use regression analysis to understand the most important factors in determining highly-selected packages, we collect seventeen package factors. Since these factors present information that developers can observe by examining online sources about the npm packages, we resort to extracting these factors from four different sources: 1) GitHub, which presents the package source code and other development activities such as issues and commits, 2) npm, which contains information about npm packages that developers can examine on the npm website, 3) *npmjs*, which is the official search engine used by the npm platform and provides metadata about the packages, and 4) Snyk, which is a service that provides a dataset of vulnerable npm packages and their versions. Table 4.4 shows the factors with their names, value types, and descriptions. In the following, we present the detailed process of extracting the studied factors from each data source:

**GitHub:** to collect the repository level factors, we use the official GraphQL API ([GitHub, 2021a](#)) to collect the number of forks, watchers, stars, and closed issues for each npm package in our dataset. Since GraphQL API does not provide direct access to the number of contributors and build status of each package repository, we use the GitHub REST



Table 4.4: List of factors values with their description.

Factor	Type	Description
Forks	Number	Forks count on GitHub
Watchers	Number	Watcher count on GitHub
Contributors	Number	Contributors count on GitHub
Downloads	Number	Downloads count from npm
Stars	Number	Stars count on GitHub
Dependencies	Number	Count of dependencies from package.json
License	Boolean	Weather has a permissive license
Documentation	Number	Size of README file
Test Code	Boolean	Whether has a test script
Build Status	Number	Percentage of failed jobs on last commit
Vulnerabilities	Number	Percentage of vulnerable versions
Badges	Number	Count of badges in the README file
Website	Boolean	Weather has a website
Releases	Number	Frequency of releases
Closed Issues	Number	Count of closed issues on GitHub
Commit Frequency	Number	Count of commits in the last year
Usage	Number	Count of dependent repositories on GitHub

API ([GitHub, 2021b](#)) to count the number of contributors, and the list of build status. In addition, to measure the commits frequency, we cloned the GitHub repositories for each of the selected packages and count the number of commits on all branches that was committed in the latest year. Finally, we wrote a web crawler to collect the package usage factor from the GitHub web interface, which presents the number of other GitHub repositories that depend on the package.

**npm:** from the official npm registry, we retrieve the list of releases for each package in our dataset. Then, we calculate the release frequency factor by dividing the number of releases by the number of days. Likewise, to present the dependencies factor, we use the registry to count the number of dependencies that a package uses in its last version. Also, to calculate the documentations factor for each package, we consider the size of the readme file. We then measure its size in terms of the number of its characters.

Also, we use both the npm registry and GitHub GraphQL API consecutively to retrieve the name of the license that a package declares. We then classify the licenses into three categories: 1) permissive licenses, 2) semi-permissive licenses, and 3) restrictive licenses (Team, 2019). Finally, we retrieve the list of badges for each package using a tool called detect-readme-badges<sup>2</sup>. Once we have the list of badges, we calculate the badges factor by counting the number of badges used by the package.

**npmjs:** for the download factor, we use the official npm search (*npmjs*<sup>3</sup>) through its API to collect the number of downloads. Next, we examine whether the package has test code to represent the test code factor. Additionally, we use the *npmjs* API to determine the website factor. To do so, we extracted the website URL for each package in our dataset. Since some packages refer to their GitHub repository as their main website, we filter out those URL addresses.

**Snyk:** to collect the vulnerabilities factor for each package in our dataset, we wrote a web crawler to collect the list of vulnerable releases from the Snyk web interface. Then, we divide the number of vulnerable releases by the total number of releases for each package to calculate the vulnerabilities factor.

Table 4.4 shows the name, value type, and description of the factors that we use to build our logistic regression models. Since some packages do not have values for some factors, we filter out these packages from our dataset. In the end, we were able to collect factor values for 1,332 highly-selected packages and 1,195 not highly-selected packages.

### 4.3.2 Analysis Method

To quantitatively examine the most impactful factors that determine highly-selected packages, we use logistic regression analysis. In our study, we examine the selected 2,427 packages, which we classified into highly-selected and not highly-selected packages. We

---

<sup>2</sup><https://www.npmjs.com/package/detect-readme-badges>

<sup>3</sup><https://npmjs.io>

then build a logistic regression to model the dependent variable, whether a package is highly-selected or not highly-selected. In the following sections, we describe the steps used to build the logistic regression model.

### **Correlation Analysis**

Since the interpretation of the logistic regression model can be affected by the highly correlated factors (Midi, Sarkar, & Rana, 2010), we first start by removing highly correlated factors in our dataset. Thus, we compute the correlation among the independent variables using Spearman's rank correlation coefficient. We used Spearman correlation because it is resilient to non-normally distributed data, which is the case for our independent variables (Kendall, 1938). We consider any pair of independent variables that have a Spearman's coefficient of more than 0.8 to be highly correlated. We selected the cutoff of 0.8 Spearman since prior work suggested and used the same threshold for software engineering data (Li, Shang, Zou, & Hassan, 2017; Tian et al., 2015). Figure 4.4 shows the hierarchical clustering based on the Spearman correlation among our independent variables. From Figure 4.4, we observe that three factors are highly correlated, which are stars, forks, and watchers. Finally, for these three factors, we only keep one factor, which is the number of stars. After this analysis, we end up having fifteen unique variables.

### **Redundancy Analysis**

Once we remove the highly correlated factors, we also apply redundancy analysis to detect variables that do not add information to the regression analysis (Harrell Jr, 2015). Thus, we remove them, so they do not affect the interpretation of our logistic regression model. In our dataset, we did not find redundant variables among the remaining fourteen factors.

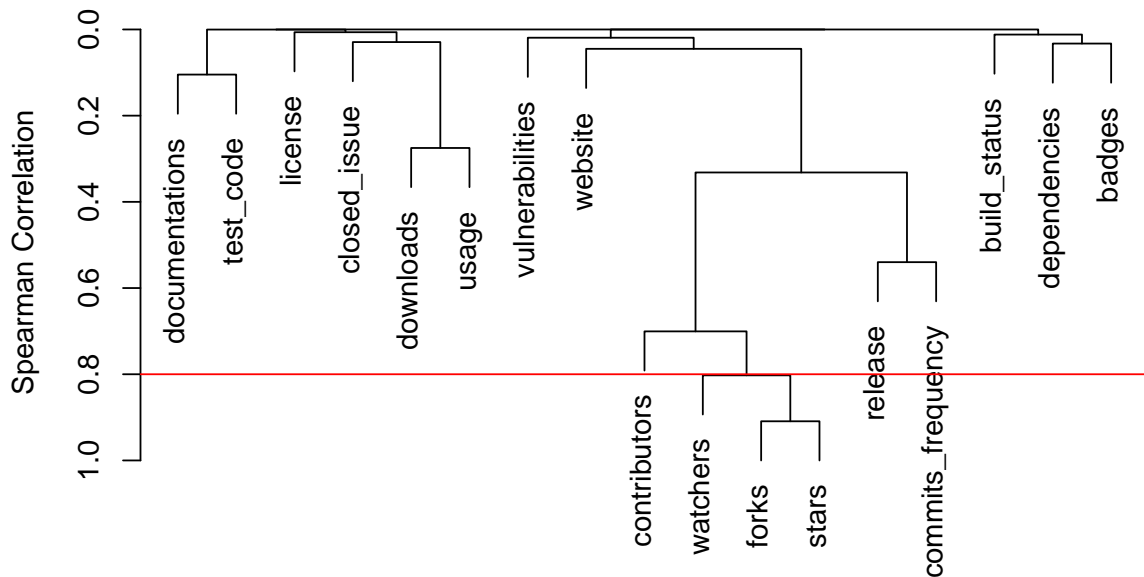


Figure 4.4: The hierarchical clustering shows the factors that present highly-selected packages. We apply the Spearman correlation test and use a cut-off value of 0.8, to eliminate highly correlated factors. This analysis left us with fifteen factors that is used in the regression analysis.

## Logistic Regression

To build our logistic regression model, we follow steps that have been applied in prior studies (Lee et al., 2020). After identifying the factors that may impact the selection of an npm package, we used logistic regression to model the highly-selected packages. Since prior studies show that using logistic regression may be affected by the estimated regression coefficient (Harrell Jr, 2015; Lee et al., 2020), we train our model using several bootstrap iterations. Similar to prior work (Lee et al., 2020), we create 100 rounds of bootstrap samples with a replacement for training and testing sets that ensure the testing samples are not included in the training set and vice versa. Then, we build a logistic regression model on the created bootstrap training samples, one for each iteration (i.e., 100 times) and test it on the testing samples. In the end, we calculate the mean of the sample statistics out of the 100 bootstrap samples.

## Evaluating Performance

Once we build our logistic regression model, we want to examine the performance of the built model. Hence, we use the area under the receiver operating characteristic curve (ROC-AUC), an evaluation measurement known for its statistical consistency. An ROC-AUC value ranges between 0 and 1, where 1 indicates perfect prediction results, and 0 indicates completely wrong predictions. Prior studies show that achieving a 0.5 ROC-AUC value indicates that the model's predictions are as good as random. However, an ROC-AUC value equal to or more than 0.7 indicates an acceptable model performance for software engineering datasets (Lessmann, Baesens, Mues, & Pietsch, 2008; Nam & Kim, 2015; Yan et al., 2019). Our logistic regression model achieved an ROC-AUC value of 0.74.

## Usage Factors Importance

To investigate which of the examined factors are the most impactful in our logistic regression modeling of highly-selected packages, we use the Wald  $\chi^2$  maximum likelihood tests value of the independent factors in our model (Harrell Jr, 2015). The higher the Wald  $\chi^2$  statistics value of an independent factor, the greater the probability that its impact is significant.

We also generate nomogram charts to present the studied factors' importance on our logistic regression model (Harrell Jr, 2015; Iasonos, Schrag, Raj, & Panageas, 2008). Nomograms are easy to explain charts that provide a way to explore the explanatory power. Since the Wald  $\chi^2$  test provides us with only the explanatory power, we use the nomogram to show us the exact interpretation of how the variation in each factor influences the outcome of the regression model. The Wald  $\chi^2$  also does not indicate whether the studied factors have positive or negative roles in determining highly-selected packages or not, while the nomogram provides such information.

Figure 4.5 shows the nomogram of the logistic regression model. The line against each

factor in the figure presents the range of values for that factor. We use the points line at the top of the figure to measure the volume of each factor contribution, while the total points line at the bottom of the figure presents the total points generated by all the factors. In our analysis, the higher the number of points assigned to a factor on the x-axis (e.g., the number of stars has 100 points), the larger its impact is on the logistic regression model.

To examine whether the difference in factor values between highly-selected and not highly-selected npm packages is statistically significant, we performed a Mann-Whitney test to compare the two distributions for each factor and determine if the difference is statistically significant, with a  $p$ -value  $< 0.05$  (Mann & Whitney, 1947). We also use Cliff's Delta ( $d$ ), which is a non-parametric effect size measure to interpret the effect size between highly-selected and not highly-selected packages. As suggested by Grissom and Kim (2005), we interpret the effect size value to be small for  $|d| < 0.33$ , medium for  $0.33 \leq |d| < 0.474$  and large for  $|d| \geq 0.474$ .

### 4.3.3 Study Results

Table 4.5 shows the values of the Wald  $\chi^2$  and the  $p$ -value for the selected fifteen factors that may impact the highly-selected npm packages. Figure 4.5 also shows the estimated effect of our factors using nomogram analysis (Iasonos et al., 2008). In addition, Table 4.6 shows the Mann-Whitney test's  $p$ -value and the effect size for each of the examined factors. Overall, we observe that the regression analysis supports the main qualitative findings. However, it controverts with the importance of some factors.

From Table 4.5, we observe that the number of downloads has the most explanatory power with a Wald  $\chi^2$  value equal to 63.00 when we model the probability of highly-selected npm packages. The second most important factor in modeling highly-selected packages is the number of stars a package has (Wald  $\chi^2 = 24.21$ ). Figure 4.5 also shows that npm packages that have a high number of downloads and received a high number of

Table 4.5: The result of our logistic regression analysis for investigating the most important factors.

<b>Factors</b>	<b>Wald <math>\chi^2</math></b>	<b><i>p</i>-value</b>	
Downloads	63.00	0.000	***
Stars	24.21	0.000	***
Closed Issue	17.62	0.000	***
Vulnerabilities	16.47	0.000	***
Badges	12.21	0.001	***
Documentation	11.61	0.001	***
Dependencies	8.04	0.005	**
Build Status	5.54	0.019	*
Test Code	4.62	0.032	*
Contributors	4.41	0.036	*
Commits Frequency	2.34	0.126	
Release	2.32	0.127	
License	1.68	0.196	
Usage	1.15	0.283	
Website	0.02	0.880	

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

stars have a high chance to be highly-selected packages. Table 4.6 shows that the Mann-Whitney test also indicates that the number of downloads and stars of highly-selected and not highly-selected packages are statistically significantly different, with a large Cliff’s delta effect size.

Our regression analysis shows that documentation and vulnerability factors also have explanatory power as well. Developers report these two factors in our survey to have a high impact when selecting npm packages. With a Wald  $\chi^2$  value equal to 16.47, packages that have a high percentage of vulnerable versions have higher impact power and the same apply for the size of the readme files with Wald  $\chi^2 = 11.61$ . In addition, Figure 4.5 confirms that documentation and vulnerabilities have a positive contribution to the probability of a npm package being highly-selected. The Mann-Whitney test also confirms that the percentage of vulnerable releases and the documentation of highly-selected and not highly-selected packages are statistically significantly different, but with negligible Cliff’s delta

Table 4.6: Highly-selected Vs. Not highly-selected packages: Mann-Whitney test ( $p$ -value) and Cliff’s Delta ( $d$ ).

<b>Factor</b>	<b><math>p</math>-value</b>	<b>Cliff’s Delta (<math>d</math>)</b>
Usage	0.000	0.814 (large)
Downloads	0.000	0.548 (large)
Stars	0.000	0.498 (large)
Contributors	0.000	0.464 (medium)
Documentation	0.000	0.128 (negligible)
Vulnerabilities	0.000	0.071 (negligible)
Website	0.000	0.069 (negligible)
Build Status	0.000	0.067 (negligible)
License	0.001	0.022 (negligible)
Test Code	0.091	0.022 (negligible)
Releases	0.000	0.278 (small)
Commit Frequency	0.000	0.244 (small)
Badges	0.000	0.223 (small)
Dependencies	0.000	0.189 (small)
Closed Issues	0.000	0.159 (small)

effect sizes.

Interestingly, our regression analysis shows two of the studied factors that have an explanatory power when they are used to model the probability of highly-selected npm packages, which are the number of badges that the package has and the number of closed issues on Github. However, our survey results show that developers tend not to consider these factors when searching for an npm package to use. Table 4.5 shows that the number of closed issues is the third most important factor with a Wald  $\chi^2$  value equal to 17.62 while the number of badges is placed fifth, having a value of Wald  $\chi^2$  equal to 12.21. Furthermore, Figure 4.5 shows the number of badges has a positive contribution with the probability that the package will be highly-selected packages. When we examine whether the difference in these factors values between highly-selected and not highly-selected packages is statistically significant, we found they are significant but with a small effect size.

In addition, our nomogram analysis shows that the number of contributors as a factor has a negative contribution to the probability of a package being highly-selected, while



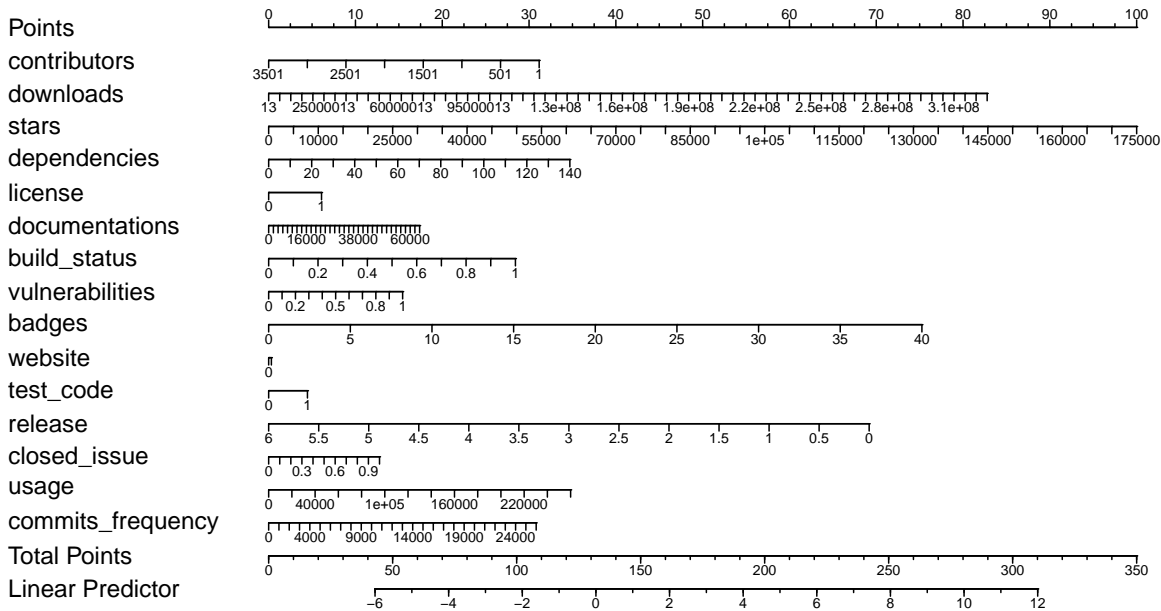


Figure 4.5: The nomogram visually presents the impact of each of the studied factors in determining highly-selected npm packages. The logistic regression model used to generate this nomogram achieved a median ROC-AUC of 0.74 on 100 out-of-sample bootstrap iterations.

the regression analysis shows that this factor has a modest explainable power (Wald  $\chi^2 = 4.41$ ).

### 4.3.4 Lifetime Analysis

Our quantitative analysis investigated the factors that make an npm package a highly-selected one. Our analysis used the cumulative number of dependent packages in the npm ecosystem as a proxy to select the highly-selected packages. Thus, the age of the packages may impact our results. For example, a young package with 100 dependent packages gained in one month tends to be more highly-selected than a package with 200 dependents gained in two years.

To examine the effect of the package lifetime on our results, we normalized the dependent factor (i.e., number of dependent packages) in our analysis by the package age (i.e., number of days since the first release of the package). We then follow the same approach

Table 4.7: The result of our logistic regression analysis for investigating the most important factors considering lifetime of the studied packages.

Factors	Wald $\chi^2$	<i>p</i> -value	Shift	
Downloads	60.66	0.000	***	-
Badges	16.83	0.000	***	↑ 3
Closed Issue	16.65	0.000	***	-
Stars	15.83	0.000	***	↓ 2
Vulnerabilities	12.63	0.000	***	↓ 1
Dependencies	8.94	0.003	**	↑ 1
Build Status	6.98	0.008	**	↑ 1
Documentation	6.80	0.009	**	↓ 2
Contributors	2.92	0.087	.	↑ 1
Test Code	2.91	0.089	.	↓ 2
Commits Frequency	1.90	0.168		-
Usage	1.54	0.214		↑ 2
License	1.53	0.217		-
Release	1.28	0.257		↓ 2
Website	0.00	0.954		-

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

as in Section 4.3.2. First, we regroup the studied packages into highly-selected and not highly-selected groups based on the number of dependent packages after normalizing the values by the age of the packages. We consider the top 20% packages as a highly-selected package and the low 20% as low-used packages. We ended up with 1,214 highly-selected and 1,213 low-used packages. Second, we apply correlation and redundancy analysis to the data. Finally, we apply the same analysis steps described in Section 4.3.2 to build our regression model. The model that we built based on the normalized dependent factor achieves ROC-AUC value of 0.74.

Table 4.7 shows the result of our logistic regression analysis for investigating the most important factors after the normalization along side with the shift in the ranking compared to the original analysis. From Table 4.7, we see that results before and after the normalization are similar. In particular, we observe that number of downloads, stars, and documentation are still among the most important factors. Overall, this analysis shows that our

observations still hold.

In summary, our quantitative analysis complements developers' perceptions about the factors that they look for when selecting an npm package to use. In particular, our results show that highly-selected npm packages tend to possess characteristics that include a high number of downloads, stars, and a higher ratio of closed issues. Lastly, in contrast to our qualitative analysis results, our regression analysis shows that a higher number of badges is an essential characteristic of highly-selected npm packages.

## 4.4 Discussion and Implications

Our study has many direct benefits for the ecosystem maintainers and the npm community, particularly package owners and developers who use the npm packages. We discuss these implications and benefits in the following.

**The npm software ecosystem maintainers should pay attention to certain aspects of the packages when building package search or recommendation tool.** Several package search tools have been proposed and deployed, which can be classified into two main categories. The first category based on keyword search ([Kashcha, 2017](#); [npm, 2017a](#); [Temple, 2017](#)). These tools are limited because they do not take into consideration the quality aspect of the packages. Tools from the second category provide package search while considering some quality aspects of the packages, e.g., the *npms* tool ([Cruz & Duarte, 2018](#)). While *npms* is the official search tool used by the official npm website, it has some limitations. The main limitation of *npms* is that it assigns different weights of the used aspects without a clear justification, which negatively affects the quality of the search engine ([Abdellatif et al., 2020](#)). Our examination of *npms*' source code and documentation shows that *npms* arbitrarily gives weights to certain aspects when ranking the packages.

We recommend that the npm ecosystem could use our results to build more robust

search tools. npm maintainers can integrate our rank of the important factors to weigh each factor's contribution when used in a search tool, which they are based on developers' perceptions.

**Several package characteristics should be carefully examined by developers when choosing an npm package to depend on in their projects.** As mentioned earlier, our results indicate that highly-selected packages possess specific characteristics. For example, our regression analysis results show that the number of closed issues in the package repository is commonly related to highly-selected packages. We believe that JavaScript developers can use our results to build systematic guidelines for choosing an npm package to use. In fact, there have been several attempts to help developers create such a guideline (Franch & Carvallo, 2003; Semeteys, 2008; Wasike, 2010). However, their main drawback is that they focus on selecting packages in a specific context or propose general guidelines to select open source components. In addition, they do not consider package characteristics that npm provide, such as the number of downloads.

**To promote their packages, the owner of npm packages should provide a clear indication of their packages' characteristics.** Gaining more popularity within the software ecosystem requires putting more effort into signaling the published packages' quality. Overall, all the package factors that our qualitative and quantitative results highlight are essential factors that package owners can employ to attract more users. For example, many responses indicate that package documentation is an important factor when looking for a package to use. Based on these findings, we recommend developers invest more effort in making their package documentation, particularly readme files, clearer and up to date.

## 4.5 Threats to Validity

In this section, we discuss the potential threats to the validity of our work.

### 4.5.1 Internal Validity

Internal validity concerns factors that could have influenced our results. To qualitatively understand the factors that may impact the use of an npm package, we survey JavaScript developers. While we carefully design our survey based on the guideline provided in (Dillman, 2011), our survey may have been influenced by some factors. First, our survey participants may poorly understand some of the factor statements. To mitigate this limitation, we conducted a pilot survey where we gave our survey to three expert JavaScript developers and incorporated their feedback about the survey. Second, we have a list of well-defined factors that may impact selecting an npm package. Even though we choose to study these factors since they are used in the literature, we may miss some other factors. To mitigate this threat, we have one open-ended question, where we ask developers to provide us with any factors that are missed in our survey (Dillman, 2011). That said, none of our survey responses report any new factors that can be quantitative.

To recruit participants in our survey, we resort to developers who publish and use packages from the npm ecosystem. At the beginning of the survey, we articulated that the purpose of our study is to understand how developers select npm packages. This description may attract more attention from developers, who use npm packages more.

### 4.5.2 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 on npm ecosystem, we used *npmjs* platform (Cruz & Duarte, 2018) to measure various quantitative factors related to testing, community interest, and download counts. Our measurements are only as accurate as *npmjs*; however, given that it is the main search tool for npm, we are confident in the *npmjs* metrics. We also use Snyk (Snyk, 2021) to calculate the number of vulnerabilities that affect the studied packages, and our measurements are as accurate as *libraries.io*. We

resort to using the Snyk data since it has been used by other prior work ([Mahmoud Alfadel, 2021](#); [Zapata et al., 2018](#)). In addition, we wrote a crawler to extract factors from the Github platform through the use of Github API, so our collected data may be affected by the accuracy of these public APIs. Furthermore, In our study, we investigate package factors that can be observed in a mechanical way (e.g., examine the Github repository of the package). However, developers may select npm packages based on a discussion or recommendation by other developers. Thus, our studied factors may not present the whole picture.

### **4.5.3 External Validity**

Threats to external validity concern the generalization of our findings. In our study, we investigate the factor that impacts highly-selected packages that are published on the npm ecosystem. Our results may not be generalized to other software ecosystems such as maven for Java or PyPi for Python. However, since npm ecosystem is the most popular software ecosystem, this gives us confidence in our results. Also, scientific literature shows that studying individual cases has significantly increased our knowledge in areas such as economics, social sciences, and software engineering ([Flyvbjerg, 2006](#)). Second, our dataset used in the quantitative analysis presents only open source packages hosted on GitHub that do not reflect proprietary packages or packages that are hosted on other platforms such as GitLab and BitBucket. Furthermore, we surveyed 118 JavaScript developers, so we do not claim that our results are generalized to other developers how do not know JavaScript or the npm software ecosystem.

Finally one criticism of empirical studies results is “I know it all along” thought or nothing new is learned. However, such common knowledge has rarely been shown to be trusted and is often quoted without scientific and research evidence. Our work provides

such evidence and supports common knowledge (e.g., “packages with good documentations tend to be highly-selected packages”) while some is challenged (e.g., “developers do not consider the number of badges when selecting a new package to use.”).

## 4.6 Chapter Summary

In this work, we use a mixed qualitative and quantitative approach to investigate the characteristic of highly-selected npm packages. We start by identifying seventeen packages selection factors based on our literature review and used by existing online package search tools. Then, we qualitatively investigate the factors developers look for when choosing an npm package by surveying 118 JavaScript developers. Second, we quantitatively examine these factors by building a logistic regression model using a dataset of 2,427 npm packages divided into highly-selected and not highly-selected packages.

Among our main findings, we highlight that JavaScript developers believe that highly-selected packages are well-document, receive a high number of stars on GitHub, have a large number of downloads, and do not suffer from security vulnerabilities. Moreover, our regression analysis complements what developers believe about highly-selected packages and shows the divergences between the developers’ perceptions and the characteristics of highly-selected packages.

Our results help in deciding how to select package dependencies. Actively maintained packages frequently receive updates to patch security vulnerabilities, fix bugs, or add new features. However, updates could introduce bugs that break the existing functionalities. Hence, in the next chapter, we propose a technique to detect breakage-inducing versions of third-party dependencies.

# Chapter 5

## An Approach to Identify Breaking Updates

An earlier version of this chapter is published In Proceedings of the 17th International Conference on Mining Software Repositories (MSR) 2020.

### 5.1 Introduction

Today's software systems are large and complex. Many of these software systems are not built from scratch, but rather leverage others' code that has been built in the past to accelerate their own development. One particular driver of this code reuse is the growing popularity of software ecosystems such as Node.js Package Manager (npm),<sup>1</sup> which provides a platform for developers to share their own and use others' code. Thus, developers commonly publish their reusable code as packages on npm, which can be used in current and future projects developed by members of the npm ecosystem (Wittern et al., 2016).

Code reuse has many advantages, including allowing software systems to be developed faster, include richer features, and even achieve higher quality (Abdalkareem et al., 2017;

---

<sup>1</sup><https://www.npmjs.com>



[Abdalkareem, Oda, et al., 2020](#)). However, this often comes at an increased cost of having to manage these dependencies ([Mirhosseini & Parnin, 2017](#)). Specifically, as the software evolves (and its dependencies do as well), updating these dependencies can become more risky ([Bogart et al., 2015](#); [Decan et al., 2017, 2019](#)).

The question of whether one should update to the newest released version is an important development decision. On the one hand, updating means that developers will get the newest features and important patches ([Cadariu et al., 2015](#); [Decan et al., 2018b](#)). On the other hand, the fear of an update breaking existing functionality often lingers on the minds of developers, making them resort to version pinning their dependencies, or other suboptimal solutions ([Decan et al., 2018a](#); [Kula et al., 2017](#); [Zerouali et al., 2018](#)).

To ensure the stability and quality of newly released dependencies, developers often run their own tests. This has proven to be a good solution and some tools (e.g., Greenkeeper<sup>2</sup>) support the automation of such approaches. However, in many cases, developers are still forced to “roll back” updates to packages because they introduce regression in their system functionality. Indeed, [Mirhosseini and Parnin \(2017\)](#) found that there is a need for new techniques to increase the confidence in automated dependency updates.

To tackle the aforementioned issues, we set out to *leverage knowledge from the crowd to provide insights about the risk of a newly released version of a package*. Specifically, we propose a technique that runs the tests of *other* projects that depend on a specific version and use their test outcome(s) as crowd-sourced indicators of the risk of adopting a newly released package.

The technique runs tests from dependent projects before and after updating a target dependency from a prior version to a newer version. Unless an update is intentionally breaking backwards compatibility (e.g., a major release), the tests from the prior version should continue to pass in the newer version ([npm Documentation, 2018](#); [Raemaekers, van](#)

---

<sup>2</sup><https://greenkeeper.io>

[Deursen, & Visser, 2017](#)).

To detect breakage-inducing versions, we execute the tests of dependent projects that depend on the prior version of the target dependency. For those tests that pass on the prior version, we re-execute them after updating the target dependency to the newer version. Tests that pass the execution on the prior version but not the execution on the newer version may indicate that the newer version has introduced a breakage.

To evaluate the proposed technique, we perform an empirical study of ten cases where an upgrade was rolled back because of a breakage-inducing version. Our study evaluates: 1) the coverage of the tests from other dependent projects and 2) the ability of the technique to indicate potential problems with a newer version of a target dependency.

We find that the tests from other dependent projects have varying test coverage, and in some cases, this coverage can be as high as 55%. Also, we find that of the 10 cases where a dependency was rolled back, tests from other dependent projects were able to indicate a failure in 60% of the time. The following are the key contributions of our work:

- We propose an approach to detect breakage-inducing versions of third-party packages by leveraging tests from “the crowd”.
- We perform an empirical study of ten cases of real word breakage-inducing versions to demonstrate the effectiveness of our approach.
- We make our dataset publicly available to facilitate further research ([Mujahid, Abdalkareem, Shihab, & McIntosh, 2019](#)).

The remainder of this chapter is structured as follows. We start by describing the background information using a motivational example in Section 5.2. Section 5.3 provides an overview of the study design. Section 5.4 presents the results of our research questions. We discuss our results in Section 5.5. Section 5.6 presents the threats to validity of our study. Finally, Section 5.7 draws conclusions.

## 5.2 Background and Motivating Example

To help illustrate how our approach works and the challenges of updating the dependencies in the context of the npm ecosystem, we provide a simple motivating example.

Amy is as a web developer that is responsible for developing and maintaining web applications for three projects in her company. Her projects depend on open source projects from npm to leverage backend and frontend functionalities for her company's applications. Each of the applications uses on average 50 npm dependencies. As with many packages on npm, the dependencies she uses get updated frequently. Amy wants to be more proactive in managing her software dependencies, so she uses Greenkeeper, a tool that automatically checks for dependency updates. If a dependency has a newer version available, Greenkeeper updates the dependency to the newer version and runs the tests that Amy wrote for her application. If the newer version passes the tests, Greenkeeper creates a pull request to update the dependency.

One day, Amy started to receive complaints from her application's users about unexpected behaviour. When she debugged the issue, she found that a recent change that she made by updating to a newer version (0.14.0) of the `cheerio` dependency introduced the issue. Even though all tests passed, the tests that Amy wrote were not able to detect the breakage behaviour in the updated dependency - the tests simply did not cover the case causing the unexpected behaviour. The immediate solution was to rollback the dependency update to the prior version (0.12.4). Even though this procedure fixes the issue, Amy starts to become concerned about breakage-inducing versions. Through a quick web search, Amy finds that she is not the only one that suffers from this breakage-inducing version problem.

Our proposed technique aims to help developers like Amy, be more confident when they update their dependencies. In this example, Amy's tests did not detect the breakage-inducing version, however, as we illustrate in Figure 5.1, Amy is not the only one that uses

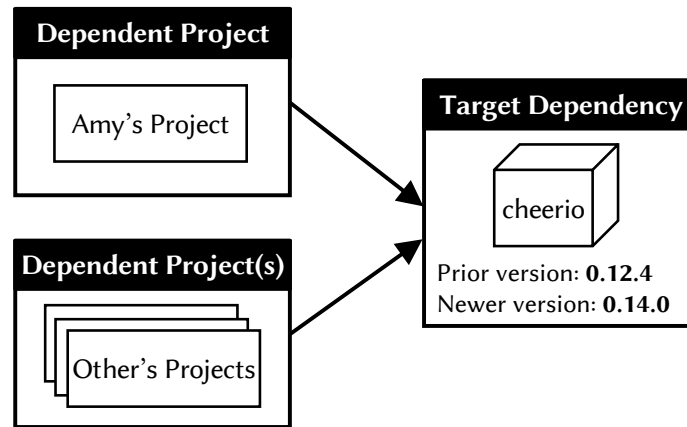


Figure 5.1: Motivating example overview and used terminology.

the target dependency. Other developers also use the same dependency in their projects. If Amy's tests failed to detect the breakage-inducing version, other's tests may have potentially caught that breakage-inducing version. When a newer version of a dependency is out, why not wait until other developers update to the newer version and based on their test results determine whether or not we should update. If the newer version breaks others' code, there is a high chance that it may break Amy's code as well.

Our technique simulates something similar, but at a very high level. Rather than waiting for others to update, we update the target dependency from the prior version to the newer version for the dependent projects that use the target dependency and check whether it breaks their tests or not. If the update breaks the tests, we flag the newer version of the target dependency as a breakage-inducing version. Even though when we mark a version as a breakage-inducing version, it may not mean it will be a breakage-inducing version for every target dependency, however it means that newer version might be risky since it broke other's tests. Hence, Amy, for example, should be careful when she wants to update to this specific newer version.

## 5.3 Study Design

In this section, we discuss the main data used in our study and its collection process.

### 5.3.1 Corpus of Candidate Packages

Since the main goal of our study is to detect breakage-inducing versions of packages in npm, we first collect a large dataset of packages that are published on npm. Even though, the main intent of packages published on npm is to be used as third-party libraries by other JavaScript projects, these packages also depend on other packages to perform their tasks. To perform our study, we retrieve the list of all packages published on npm through its registry ([npm Documentation, 2019](#)). We were able to collect a total of 664,204 of npm packages as March 29th, 2018.

We choose to study packages on npm that are written in JavaScript since 1) we manually examine the source code changes of some packages and to give us confidence, we choose a programming language that the authors have expertise in, 2) JavaScript is one of the most popular programming languages on GitHub and also npm the most growing packages management systems in recent years ([Decan et al., 2019](#); [Vasilescu, Yu, Wang, Devanbu, & Filkov, 2015](#)). In addition, npm has a well structured software ecosystem with a large amount of packages.

That said, it is essential to highlight that our approach is not language or platform dependent and can be applied on dependencies written in any languages and published on any dependency ecosystem. Figure 5.2 illustrates the steps used to build our data corpus. We describe each step in more detail next.

**Apply Data Filtering (Step 1).** After obtaining the list of 664,204 packages, we want to analyze the commit history and then use the packages' tests to detect the breakage-inducing versions. However, the suggested/common practice on npm is to exclude the tests

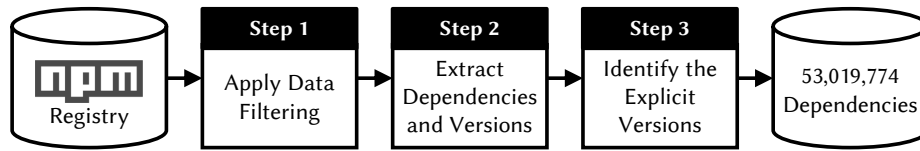


Figure 5.2: Data collection overview

and non-production code files from the published packages ([npm Documentations, 2018](#)). To recover the missed data, we rely on the `git` repositories of the packages to retrieve their test code and their development history. Thus, we filter out packages that do not have a `git` repository. We found 391,553 `npm` packages in our dataset that have a valid link to their GitHub repository.

To eliminate immature and dummy packages, we filtered out packages that have less than two commits that touch the `package.json` file, which is the dependency configuration file. It is worth mentioning that this filtering process is important to allow us to keep only packages that do update their dependencies. After applying this filter, we were left with 290,417 repositories to analyze and use as our set of dependent projects.

Once we obtain the lists of 290,417 GitHub repositories, we extracted their dependencies and tracked all changes that the developers performed on their dependency versions. Specifically, we tracked all commits that touch the package configuration file (`package.json`), which we explain next.

**Extract Dependencies and Versions (Step 2).** Since our approach relies on identifying dependent projects to test the candidate update of a dependency, we need to identify the dependencies of the dependent projects. However, dependencies and versions can change across the history of a project. Thus, we want to collect these changes for two reasons: 1) to extract dependency downgrade cases, which indicate problematic updates i.e., breakage-inducing versions, and 2) to build a precise dependency graph between the dependent projects and the dependencies based on different points in the history, which will be used later to select the dependent projects.

In order to extract the dependencies and their versions across the history of a project, we analyze all commits that touch the `package.json` file, which is a file that npm use it to recognize the package dependencies and its versions. We use the GitHub GraphQL API<sup>3</sup> to collect all commits that touch the `packages.json` file for each project in our dataset. As a result, we collected 4,200,936 commits that touch the package configuration file. On each commit, we retrieve two versions of the `packages.json` file, one showing the file before the commit ( $File^P$ ) and the other showing the file after the commit ( $File^C$ ). We parse the files and extract the dependency list from  $File^C$ . For each dependency in  $File^C$ , we extract its version from  $File^C$  and  $File^P$ . At the end of this process, we were able to extract more than 53,019,774 dependency records across the history of the projects.

**Identify the Explicit Versions (Step 3).** Developers of JavaScript projects usually do not specify the explicit version number for each of their dependencies. Instead, it is popular to use version ranges for their dependencies. Hence, we cannot link these dependency ranges to a specific version. In such cases, it is not possible to pin point the exact dependency version that was used. For example, if the range is `1.2.x` and the latest version is `1.2.1`, npm will point the dependency to this version. Later, if a newer version, e.g., `1.2.2`, is released, npm will point the dependency to it, and so forth. npm ensures that the updates respect the version ranges specified by developers. For example, if the newer version is `1.3.0`, then npm will not update to it since it does not satisfy the specified version range `1.2.x`.

Thus, to identify the version of a dependency that was used to satisfy a version range, we map version ranges to the latest satisfied version that is released before the date of the commit that introduces  $File^C$ . In order to perform this step, we: 1) replicated the npm registry locally; and 2) built a registry proxy that takes the commit date as an argument and simulates the registry as if it was at that specific date. Then, we use the built proxy to

---

<sup>3</sup><https://developer.github.com/v4/>

Table 5.1: The Selected Ten Downgrading Cases.

Package	Downgraded			
	From		To	
ESLint	2.4.0	(^2.2.0)	2.2.0	(~2.2.0)
Express	3.4.0		3.3.4	
Express	4.2.0	(^4.0.0)	3.4.8	(~3.4.x)
Intl.js	1.2.5	(^1.2.4)	1.2.4	
jQuery	2.2.0	(^2.1.1)	2.1.4	(~2.1.4)
Marked	0.3.5	(^0.3.2)	0.3.3	
Marked	0.3.9	(^0.3.6)	0.3.7	
Nodemon	1.12.1	(^1.11.0)	1.11.0	
Passport	0.3.0	(^0.3.0)	0.2.0	
Request	2.83.0	(^2.53.0)	2.81.0	

intercept the result from the npm registry and remove versions that are newer (i.e., come after the date of the analyzed snapshot). The proxy helps us simulate the status of npm result at the snapshot time (commit date). Using this approach, we were able to determine the exact version of each dependency, which we later use to determine dependency downgrades and provide us with a precise list of dependent projects.

### 5.3.2 Selection of Case Studies

In order to examine the practicality of our proposed approach, we want to extract a baseline of breakage-inducing versions. Since the normal behaviour is upgrading the dependencies, a dependency downgrade can be a perfect indicator of unusual behaviour i.e., upgrades that break the tests of the dependent projects. Thus, in this study, we resort to use the downgraded cases to select our studied breakage-inducing versions that have cases of breakage-inducing versions

For every commit in our dataset, we compare the explicit versions for the ranges extracted from  $File^P$  and  $File^C$ . If the explicit version of the dependency on  $File^P$  is greater than the explicit version on  $File^C$ , we consider this as a downgrade case. We were able



to identify 9,046 possible breakage-inducing versions from 3,255 npm dependency packages by analyzing the commits from their dependent projects and detect dependency downgrades.

To isolate the downgrade behaviour from other changes, we only kept commits that do not perform any other changes besides the dependency downgrade change. In other words, we select downgrade cases where the commit only changes one line, which is the line that changes the dependency version. By adding this constraint, we were left with 1,880 possible breakage-inducing versions from 909 npm dependency packages.

In addition, to make sure the process correctly identifies cases of downgrade versions, the first two authors also performed a sanity check of randomly selected 100 downgrading commits by checking the commit messages and examining the `packages.json`. In all cases, the commit messages confirmed our results that the commits were only downgrading the dependencies.

Since the number of identified packages is a large number and it does not make sense to examine all of these cases, we decide to focus our analysis on cases that we can manage to analyze manually and perform an in-depth analysis. To evaluate our proposed approach using different real-world npm dependencies and breakage-inducing versions, we randomly selected ten downgrade cases to be used as the baseline in the evaluation of our proposed technique. In the selected cases, we consider the prior versions that the developers downgraded from as the breakage-inducing versions and the newer versions that they downgrade to as the stable versions.

Table 5.1 presents the ten randomly selected cases. The second column shows the breakage-inducing versions that the developers downgrade from as it was specified in *File<sup>P</sup>* (version ranges is shown in brackets). In the third column, the table shows the versions that the developer downgrade to, as specified in *File<sup>C</sup>* (version ranges is shown in

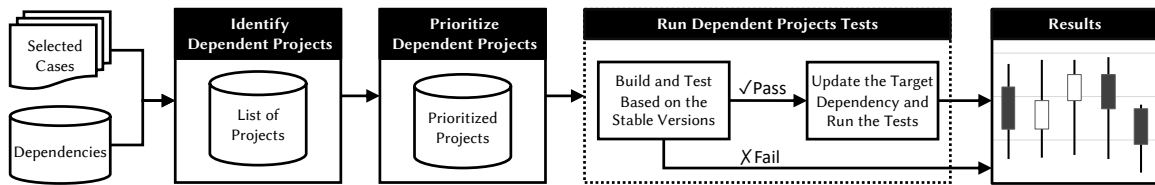


Figure 5.3: The approach overview.

brackets). Table 5.1 shows that the selected cases belong to eight npm dependency packages that are well-known and commonly used within the JavaScript developer community.

### 5.3.3 Detection of Breakage-Inducing Versions

To detect a breakage-inducing version, we rely on running the tests of projects that depend on the prior version before and after updating the target dependency to the newer version. We argue that the tests of the dependent projects can reveal the breakage-inducing versions. To do that, we propose an approach that is composed of three main steps that include, 1) identify the projects that use the prior version of the target dependency, 2) prioritize the dependent projects to run sufficient tests, and 3) automatically run the tests of dependent projects. Figure 5.3 presents our proposed approach and next, we explain these steps in more details.

**Identify Dependent Projects.** To identify dependent projects that have candidate tests for our selected ten breakage-inducing versions, we use the records of explicit package dependencies that we explained earlier in Section 5.3.1. To checkout the code on a specific point in history, i.e., where it depended on a prior version, we retrieve all commits that point to that prior version. In most cases, a project’s repository history can have several commits that point to the same version. In such case, we choose the first commit that introduces the prior version. Then, we checkout the work directory based on that commit. We were able to find 5,853 dependent projects that use the prior version of the selected cases. Finally, we want to exclude projects that do not have tests. To do so, we examine the

the `package.json` file of each project and check whether it specifies a test script. This process left us with with 3,473 dependent projects that expose test scripts. Later, we use these scripts to run the the tests.

**Prioritize Dependent Projects.** The number of dependent projects can scale to thousands of projects. Building all of them may add no value. Therefore, in practice, a budget of number of builds needs to be specified, which will impact how many dependent projects projects one can consider. To include the most valuable dependent projects, for each breakage-inducing version in our case studies, we order its dependent projects in a queue based on their test coverage percentage. To retrieve the test coverage of the dependent projects, we rely on the API of the npm search engine (*npm*s).<sup>4</sup> If more than one package has the same test coverage percentage, we prioritize the package that has a higher ranking score in *npm*s. The *npm*s scores are based on quality, maintenance and popularity - more details about how these scores are calculated can be found on *npm*s ([npm](#)s, 2016).

**Run Dependent Project's Tests.** To detect breakage-inducing versions, we need to build the dependent projects, which include installing their dependencies and running their tests. To perform this process, we build the dependent projects in an isolated environment using Docker containers. Our implementation keeps a record of the output for every stage, the time that each stage spent and the detailed test coverage reports. We achieve this by performing the following.

First, for each prior version of our selected cases, we build and run tests of its dependent projects. Our proposed approach relies on builds and tests of dependent projects that pass the prior version. The build requires installing the dependencies. However, the fact that developers can specify version ranges can be an additional point of failure. For example, a dependency could have a newer version that break backward compatibility. If a newer version satisfies the specified version range, our build will install the newer version which

---

<sup>4</sup><https://npm.io>

is incompatible. To mitigate the problem, we used the registry proxy that we implement (Section 5.3.1) to emulate the registry as it was on the commit date of *File<sup>C</sup>*. In this case, our replayed build(s) will install the version or versions that were available before the commit date. The dependent projects whose tests are already failing on the stable version are not useful since they do not provide useful information (and would not provide useful information in a real-life scenario). Therefore, we exclude dependent projects whose builds fail on the prior version. In our case study, we use a budget of 80 successful builds to be the limit. This means that if a target dependency passed tests of 80 dependent projects, we stop running more test and flag its newer version as non breakage-inducing version. At this budget, we built 1,447 dependent projects from the 3,473 projects in our dataset. Out of all builds we were able to successfully build 904 cases.

Second, for dependent projects that passed the previous building and testing stage, we update their prior version of the target dependency to the newer version. We run the same tests from the dependent projects based on the newer version and save the result. Then, we examine the saved results and if a test failed after updating the target dependency from the prior version to the newer version, we flag that version as a breakage-inducing version. This is meant to be reported to developers in an effort to help them adopt a more data-driven decision about updating to a newer version of their dependencies.

## 5.4 Case Study Results

In this section, we present the results of our empirical study with respect to our two research questions. For each research question, we present our motivation, approach, results and implications.

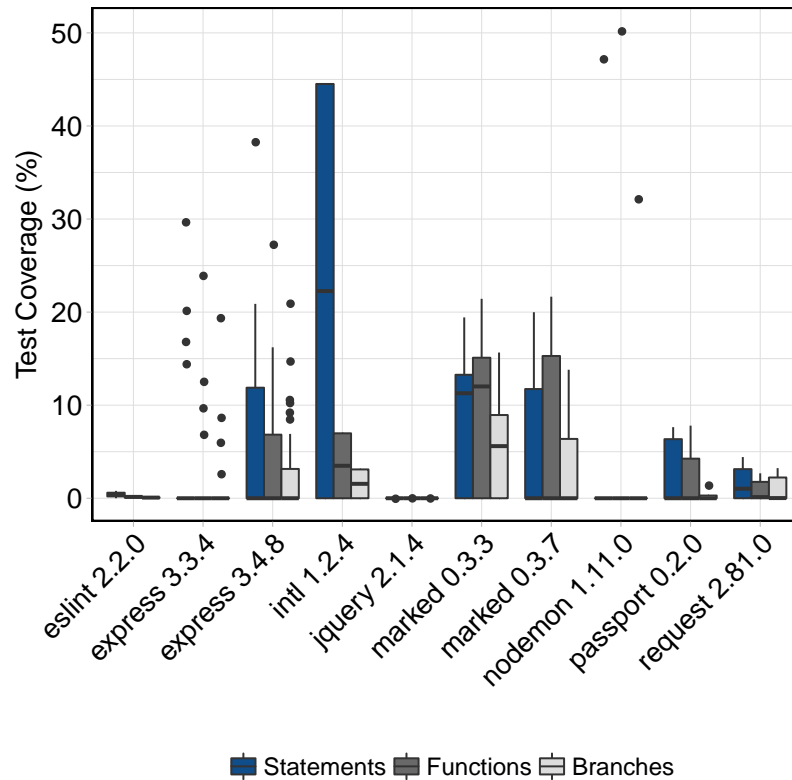


Figure 5.4: The distribution of test coverage for the studied cases based on running the tests of their dependent projects.

### **RQ1. Do the tests of dependent projects complement each other in terms of coverage?**

**Motivation.** Previous work showed that JavaScript tests tend to have low coverage (Fard & Mesbah, 2017). Therefore, we would like to know if better test coverage can be achieved by considering the tests of dependent projects. Achieving improved coverage using such tests would suggest that our technique has the potential to detect additional breakage-inducing versions. In this research question, we examine whether the tests of the dependent projects contribute to improving the test coverage of a package that they depend on.

**Approach.** To answer this question, we use an approach that depends on measuring how the tests of dependent projects contribute to cover a target dependency. We use the

Istanbul<sup>5</sup> tool to measure test coverage. We configure the tool to track the test execution for the target dependency. After running all the tests of the dependent projects, we collect the detailed test coverage reports. Then we aggregate the test report based on the paths of the covered files, including the percentages of statements, branches, functions, and lines. In cases where the same file is covered by test code of more than one dependent project, we aggregate based on the coverage map of the file, i.e., we check the coverage map for each statement, branch, and function, if an element is covered in one report but not the other, we consider it as a covered element, so we count it only once. Finally, we measure the increase in test coverage based on the order of dependent projects that we produce using the prioritization described in Section 5.3.1.

**Results.** Figure 5.4 shows the distribution of test coverage for each selected case based on statements, functions and branch test coverage. The tests of dependent projects individually covered, on median, up to 22% statement test coverage of the target dependency’s code. However, in one particular case (`nodemon 1.11.0`), we found one dependent project that covers approximately 50% of its code. Figure 5.4 also shows that there is a case (`jquery 2.1.4`) where crowd-based testing does not improve test coverage.

We also examine the effect of the number of dependent projects on the amount of test coverage that they provide. Figure 5.5 shows the cumulative statement test coverage achieved by running the dependent projects’ tests. Overall, we observe that adding more dependent projects increases the statement test coverage of the target dependency. Figure 5.5 shows that eight dependencies have an increase in the test coverage when we increase the number of dependent projects. However, the trend of statement test coverage of most of the dependencies remains stable after running the tests of approximately 20 dependent projects.

---

<sup>5</sup><https://istanbul.js.org>

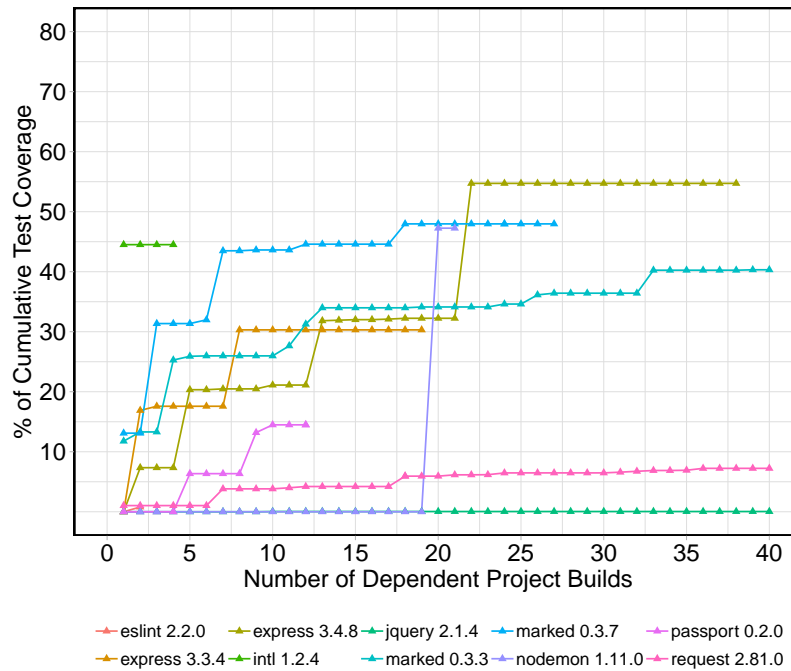


Figure 5.5: The cumulative statement test coverage for the selected ten cases based on running the testes of their dependent projects.

Moreover, we examine the degree to which test coverage is improved by adding crowd-based test results. Figure 5.6 shows the cumulative statement test coverage when the tests of the target dependency itself and the dependent projects’ tests are combined. Overall, we see that in the majority of the cases (9 out of 10) there is an improvement in the statement test coverage. In fact, in some cases, the cumulative coverage reaches as high as approximately 80%. However, in one specific case (`jquery 2.1.4`), we see that there is no improvement, we investigate the case and we found that to run its tests, we need to setup a local server that supports PHP ([jQuery Foundation, 2019](#)).

It is important to note that the coverage in this *RQ* is measured in terms of the covered statements. In addition, we also compared the percentage of covered statements, branches and functions and we did not observe a noticeable difference between them.

**Implications.** The results show that the tests of dependent projects individually and cumulatively can cover the target dependencies up to 47% and 55%, respectively. These results

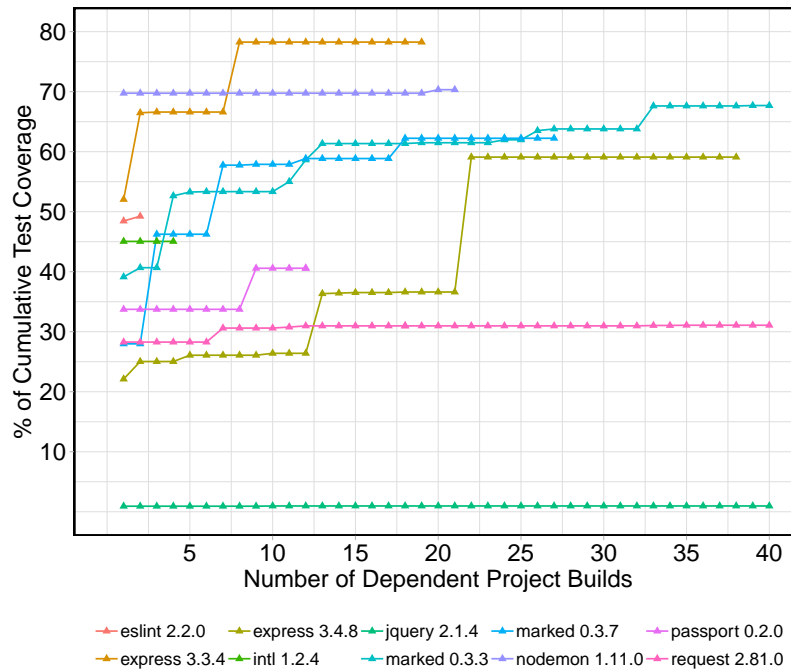


Figure 5.6: The cumulative statement test coverage for the selected ten cases based on running the testes of their dependent projects companied with their own tests.

highlight the importance of using the dependent projects to improve the capacity for detection of breakage-inducing versions. Such an approach could also improve test generation tools to produce more effective tests in the context of the ecosystem dependency network.

The dependent projects' tests can individually cover 23% on median and up to 47% of the code for the target dependency. However, leveraging the tests of the dependent projects can cover up to 55% of target dependency code.

## RQ2. How effectively can the proposed technique detect real-world breakage-inducing versions?

**Motivation.** In the previous research question, we found that dependent projects can provide tests that cover up to 55% of target dependency code. In this research question, we



Table 5.2: Builds and Tests Summary.

Cases	Number of Builds	Successful Builds (%)	Passed First Tests (#)	Passed First Tests (%)	(%) Failed After the Dependency Update	Caught as Risky Version
<code>eslint 2.2.0</code>	34	61.8	2	9.5	0	No
<code>express 3.3.4</code>	63	54.0	19	55.9	15.8	Yes
<code>express 3.4.8</code>	196	45.9	38	42.7	5.3	Yes
<code>intl 1.2.4</code>	19	63.2	4	33.3	0	No
<code>jquery 0.3.3</code>	364	37.9	42	30.4	0	No
<code>marked 0.3.3</code>	214	67.8	64	44.1	4.7	Yes
<code>marked 0.3.7</code>	73	83.6	28	45.9	7.1	Yes
<code>nodemon 1.11.0</code>	98	77.6	21	27.6	0	No
<code>passport 0.2.0</code>	74	52.7	13	33.3	7.6	Yes
<code>request 2.81.0</code>	312	92.6	83	28.2	2.4	Yes
<b>All</b>	<b>1,447</b>	<b>62.4</b>	<b>314</b>	<b>34.7</b>	<b>4.1</b>	<b>60%</b>

set out to see if using tests provided by dependent projects can catch real-world breakage-inducing versions.

**Approach.** To examine the effectiveness of the proposed approach in detecting breakage-inducing versions, we perform an experiment using the ten studied examples of breakage-inducing versions that are shown in Table 5.1. For each case, we build and run the tests of dependent projects using the prior version and once again based on the newer version using our approach described in Section 5.3.3. Cases where tests pass on the prior version(s), and have at least one failure on the newer version are flagged as breakage-inducing versions.

**Results.** The build results for all cases are shown in Table 5.2. Of the 904 successful builds, 314 builds passed the tests on the prior version. For each case, the second column shows the number of builds from dependent projects that proceed to the building stage. The third column shows the percentage of them that have a successful build, which range between 37.9% and 93.6%. In the fourth and fifth columns, we present the count and the percentage of dependent projects that passed the tests before updating the target dependency. Finally, the sixth column shows the percentage of dependent projects that failed the tests after updating the target dependency, which we use in the seventh column to flag if the newer version is a breakage-inducing version or not.

Table 5.2 shows that our proposed technique detects six of the ten studied breakage-inducing versions. For the four cases that our techniques failed to detect, we performed a manual investigation to gain insight into the reason why our proposed technique was not able to detect these cases.

**eslint 2.2.0:** ESLint is a tool that is used to identify and report problematic patterns or code that does not adhere to style guidelines in JavaScript code (Zakas, 2018). The tool is mainly used as a development dependency, which is not used in the production code. In our approach, we select the dependent projects based on their production dependencies. As a result, we were left with a small number of dependent projects.

In addition, the dependency package has the lowest percentage (9.5%) of passed tests from its dependent projects in the first testing stage. Out of the 19 dependent projects that had their tests fail, 16 of them produce the following error (`Cannot find module 'internal/fs'`). This error has a known workaround in the Node.js community. Applying this workaround may have helped, if it was known and applied in advance. Thus, we were left with only two dependent projects to test the update based on. Unfortunately, these two dependent projects only improve coverage by 0.8%, and thus, are unlikely to detect the breakage-inducing version.

**intl 1.2.4:** Intl.js is a package with five years of development history. The package is mostly used in client-side web applications to support legacy web browsers. Since web applications are not reusable dependencies by themselves, developers usually do not publish them on npm. Since we only used the dependent projects that are published on npm without considering dependent projects outside npm, we could only find 19 dependent projects for the target dependency version. Out of the 19 dependent projects, only four of them had their tests pass on the prior version before updating to the newer version. Including dependent projects in addition to the ones from npm (e.g., GitHub or Bitbucket) can help to increase the population for this case. We plan to investigate this in future work.

**jquery 2.1.4:** jQuery is a JavaScript library designed to simplify the client-side scripting of HTML. The package is mainly used to manipulate the Document Object Model (DOM) in web browser environments. Previous work showed that the DOM makes it hard for developers to test effectively (Artzi, Dolby, Jensen, Møller, & Tip, 2011; Fard, Mesbah, & Wohlstadter, 2015; Mirshokraie, Mesbah, & Pattabiraman, 2015). Our result confirms the finding of a previous study by Fard and Mesbah (2017), which shows that DOM-related tests lack proper coverage. In the case of `jquery 2.1.4`, the library test suite itself does not achieve any test coverage and also the dependent projects do not improve test coverage. This is due to a missing configuration setup (see RQ1). Hence, our approach cannot detect any breakage-inducing versions that is not covered by the test suites of the dependent projects.

**nodemon 1.11.0:** This case has 21 dependent projects that passed the tests of the prior version. However, all of them also passed the tests after updating the newer version. We investigated the case to figure out why our approach did not flag the case as a breakage-inducing version. By checking the commit messages for changes that the developers downgrade from the newer version (*nodemon 1.12.1*) to the prior version (*nodemon 1.11.0*), we find that developers downgrade to the prior version to maintain backward compatibility with older JavaScript standards (*ECMAScript 5* (Sebestyen, 2009)). The following quote is an example of a commit message.

*“Restrict version to pre-1.12 as it includes a dep requiring const”* (Brierton, 2017)

In other words, the newer version of the target dependency depends on a language feature that is not available in older JavaScript standards (*ECMAScript 5*). In our experimental setting, we only use the latest version of Node.js. Our experiment runs on the *ECMAScript 6*. Hence, downgrades that are performed due to incompatibility with *ECMAScript 6* cannot be detected. Note that this is a limitation of our experimental setup and not our approach.

In theory, if one were to extend the experimental configuration to include *ECMAScript 5* environments, our approach would detect such cases.

**Implications.** With respect to the mentioned limitations, the results show that our technique is capable of detecting breakage-inducing versions in six of the ten real-world examples. The developers of both of the dependent projects and the dependencies themselves can benefit from our technique. Developers of dependent projects can use the approach to examine their dependency versions before applying the updates. Similarly, dependency developers can use the approach to check if version updates are likely to introduce regression into their codebases.

The proposed approach was able to detect six of ten real-world breakage-inducing versions. However, our technique needs to have enough dependent projects.

## 5.5 Discussion

In this section, we discuss various aspects of our technique and how they might impact the technique's outcomes.

### 5.5.1 Technique Scalability

The first research question suggests that using more dependent projects to test a target dependency can extend the test coverage of the target dependency, which increases the chance to detect breakage-inducing versions. However, running these test cases, especially when there is a large number of dependent projects, can introduce a large overhead.

To investigate the scalability of our proposed technique, we perform an analysis to understand the time required to run the tests. To do so, we calculate the time required for tests of each dependent project and compare it to the percentage of dependent projects that

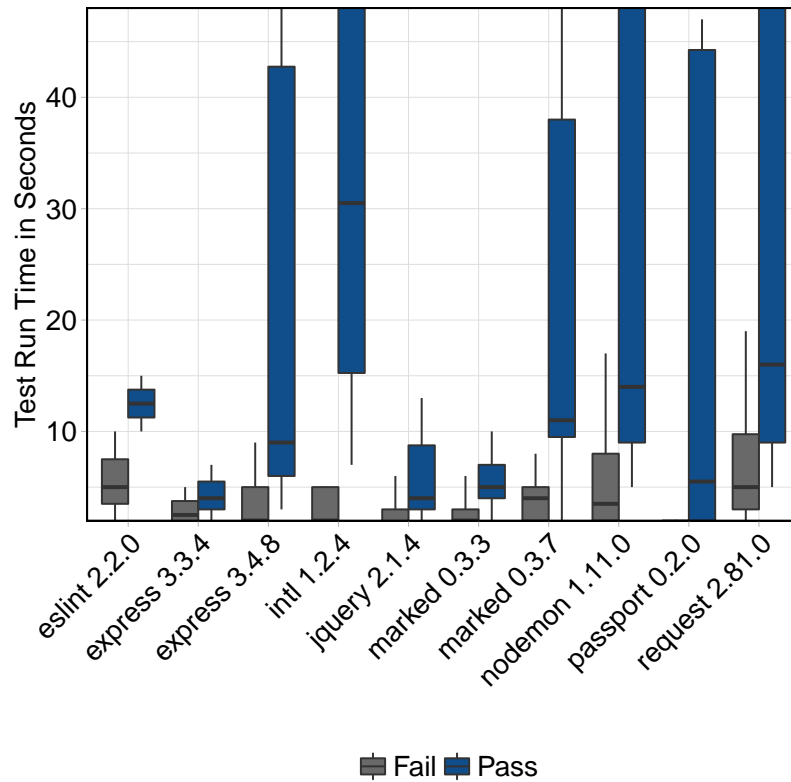


Figure 5.7: The distribution of the time that tests consume to pass or fail the builds.

we built. Figure 5.7 shows the distribution of time consumed to pass or fail the builds in our case study. We observe that the majority of passed builds consume more time than failed ones. Also, Figure 5.7 shows that the consumed time when the build pass is 66 seconds on average (median = 9).

Moreover, Figure 5.8 shows the accumulation of builds over time. Based on this, we observe that 90% of the builds consume less than 50 seconds. However, some builds consume over 890 seconds. Thus, setting a time limit for running builds can reduce the overall consumed time. For example, in our cases study, considering a time limit of 50 seconds of the total time, we can build 90% the candidate cases.

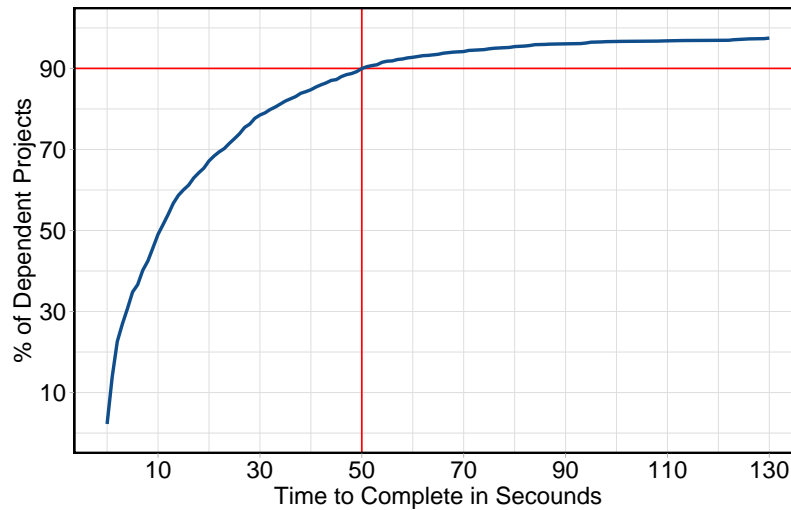


Figure 5.8: Time in seconds to the cumulative percentage of dependent projects that completed their tests.

### 5.5.2 Failed Builds

The results in Section 5.4 show that we were not able to build 37.6% of the candidate builds. Thus, we want to investigate the reasons behind the failed builds. To do so, the first two authors reviewed the logs of the failed builds and setup four classification categories. Then, they manually classified all logs and extracted the main error message. Based on the manual classification of each build log, we wrote specific regular expression to ignore the variable parts of the error messages. Then, we executed the regular expression to catch similar builds failures and classify them.

The result of the classification process is shown in Figure 5.9. In total we classify 543 failed builds. The most common reason (40.8%) is the failing in satisfying the dependencies. The next more frequent reason (22.2%) is missing a JavaScript environment requirement. For example, some projects depend on `Yarn`<sup>6</sup>, which is a dependency manager that uses the npm registry to retrieve the dependencies. For such cases our setup fails to build and run the tests successfully.

In future work we are planning to mitigate some of these issues by considering the build

---

<sup>6</sup><https://yarnpkg.com>

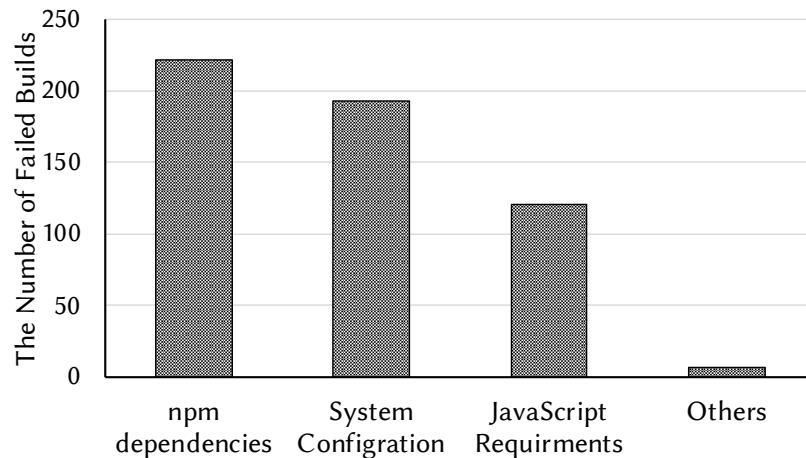


Figure 5.9: The classification of the failed builds

configuration of the continous integration systems, if possible. For example, some of the projects use `Travis CI`<sup>7</sup>, such projects include a configuration that specify pre steps and environment requirements for the build. For such cases we could satisfy the missed build requirements and increase the successful build percentage.

## 5.6 Threats to Validity

In this section, we disuses threats to validity that might influence our study.

### 5.6.1 Threats to Internal Validity

Internal validity concerns factors that could have influenced our analysis and findings. First, to evaluate our technique, we select a sample of downgraded cases to be examined. However, downgrading the dependency can be triggered for many different reasons. Therefore the results can be affected by introducing invalid evaluation cases. To mitigate this threat, we have a restricted approach to select these cases 1) we selected cases where the commits perform only one specific change, which downgrades the dependency version and 2) we make sure that the commit message of the selected cases mentions that a dependency

<sup>7</sup><https://travis-ci.org>

downgrade as the main reason for the change. Second, we randomly selected only ten cases of breakage-inducing changes. Even though this number seems to be modest, our analysis shows that using these cases we were able to systematically evaluate the practicality of our technique. Also, in the future we plan to perform a large-scale study considering the lessons learned from our current experiences.

We only use download measurement to prioritize the selected dependent projects. Other measurements could have been used, such as number of stars for the project. That said, we believe the selection of our measurement is right since it gives us a clear indication of the quality of the dependent projects (low quality projects will probably not be downloaded much). Finally, to measure the test coverage that our technique achieves through running the test from dependent projects, we use the `Istanbul` tool. Thus, our analysis heavily relies on the accuracy of the `Istanbul` tool. That said, its popularity and common usage gives confidence in our results.

In our experimental evaluation we examine ten cases and our technique was able to catch 60% of the breakage-inducing versions. This result is highly dependent on the building of dependent projects.

## 5.6.2 Threats to External Validity

Threats to external validity concern the generalization of our technique and findings. In our study, we only examine packages and dependent projects mainly written in JavaScript. Thus our findings may not generalize to other programming languages. We also examine packages published on the npm package manager and hosted on Github. However, using other dependency ecosystems may provide different result.

To prioritize the selected dependent projects that we use their tests, we rely on the measurement (i.e., number of downloads) provided by *npm*s. Thus, our prioritization is heavily impacted by *npm*s. That said, since *npm*s is the main search engine for npm and its



data has been used in prior work (e.g., ([Abdalkareem et al., 2017](#))), these reasons gives us confidence in the data provided by *npm*s.

## 5.7 Chapter Summary

Updating dependencies is an essential part of any software project. However, in open source ecosystems, where anybody can contribute by publishing reusable packages, the risk associated with updating dependencies could be problematic ([Decan et al., 2017](#)). Previous work has shown that managing dependencies is one of the most cited drawbacks of using *npm* packages ([Abdalkareem et al., 2017](#)). In this work, we propose a technique to detect breakage-inducing versions of third-party dependencies. The technique leverages tests from dependent projects to warn software teams about breakage-inducing versions. We evaluate our technique through an empirical study of 391,553 *npm* packages. We use the dependency network from these packages to identify candidate tests. We find that our proposed technique can detect six of the ten studied breakage-inducing versions. However, we can perform better if we include more dependent projects.

This chapter encourages developers to update their dependencies by empowering them with more insight into newly released versions. However, keeping healthy dependencies by updating to the latest versions of the package dependency is a good practice as long as the package is maintained. However, packages can decline and become less maintained, obsolete, or even deprecated. Hence, in the next chapter, we will mitigate this issue by proposing an approach to discover these packages as early as possible.

# Chapter 6

## An Approach to Identify Packages in Decline

An earlier version of this chapter is accepted to be published in the *IEEE Transactions on Engineering Management Journal (TEM)* **2021**.

### 6.1 Introduction

Depending on packages from software ecosystems can boost development productivity ([Abdalkareem et al., 2017](#)), and improve software quality ([Zerouali & Mens, 2017](#)). However, large size and rapid evolution of these ecosystems has its downsides as well. For example, new (and often better) packages are continuously being introduced ([Abdalkareem, Oda, et al., 2020](#); [den Besten et al., 2020](#); [Kula et al., 2017](#); [Wittern et al., 2016](#)), making other, once popular packages, obsolete, dormant or even deprecated ([Valiev et al., 2018](#)). As such, it is becoming increasingly important for application developers to ensure that they choose the right packages from the ecosystem.

Although prior work examined projects that are unmaintained ([Coelho et al., 2020, 2018](#)), to the best of our knowledge, little attention has focused on identifying packages

that lose popularity over time (i.e., are in decline). At the same time, current popularity metrics that are commonly used by developers to select packages, such as downloads and stars, are not adequate to capture a shift in community interest. For example, the number of downloads represents not only the number of times a package is installed on its own, but also the number of times it is installed as a dependency of other packages. Hence, the popularity of a dependent package could heavily impact the number of downloads of its dependencies (Dey & Mockus, 2018). Also, the number of stars a package is linked to its repository, which may include many other packages and is unlikely to decrease to reflect interest shift over time (Borges & Valente, 2018; Zhou, Vasilescu, & Kästner, 2019).

Therefore, in this chapter we use the package's centrality as a proxy of community interest. Community interest drives packages to evolve, i.e., include better features driven by community needs, keep up the package maintenance by reporting bugs to maintainers, motivate maintainers to continue supporting the package, and some times even financially support the maintainers on platforms such GitHub Sponsors,<sup>1</sup> Open Collective,<sup>2</sup> and Tidelift.<sup>3</sup> On the other hand, packages that are declining in community interest are reused less over time, may become less actively maintained, and in more extreme cases, even become abandoned (Khondhu, Capiluppi, & Stol, 2013; Valiev et al., 2018). Furthermore, the decline in community interest of a package may indicate that a better solution is drawing attention in the ecosystem, and developers are migrating to a package that better suits their needs.

Hence, our aim is to effectively identify packages that may be in decline. To do so, we use the package centrality to identify declining community interest. By definition, centrality is a measure of the prominence or importance of a node in a social network (Wasserman & Faust, 1994). Centrality has been used in many fields e.g., in finance to measure the

---

<sup>1</sup><https://github.com/sponsors>

<sup>2</sup><https://opencollective.com>

<sup>3</sup><https://tidelift.com>

stability of banks in financial networks (J. Wang, Guo, & Szeto, 2017), in electrical engineering to rank the importance of components in network infrastructures (Cadini, Zio, & Petrescu, 2009; Stergiopoulos, Theocharidou, Kotzanikolaou, & Gritzalis, 2015), and other fields including computer science and software engineering (Hong, Kim, & Haqiq, 2014; Maharani, Adiwijaya, & Gozali, 2014). In our context, centrality allows us to rank the packages based on the popularity/importance of packages that depend on them. Specifically, we use the PageRank algorithm to evaluate the shift in their centrality over time. The intuition is that packages that have a consistent decrease in the centrality ranking are likely to be packages in decline. Hence, package developers should be careful when depending on such packages.

The popularity and scale of the npm ecosystem makes it an ideal candidate for our study. We evaluate the effectiveness of using package centrality in identifying npm packages that are in decline. We formalize our study through the following research questions:

- RQ1: How effective is our approach in detecting packages that are in decline?
- RQ2: How early can our approach detect packages that are in decline?
- RQ3: How does our approach compare to other metrics in detecting packages that are in decline?

Our findings show that our approach can detect 87% of packages in decline with high accuracy, on average 18 months before current popularity metrics show the decline. Also, we find that our approach can detect packages in decline more than 16 months (on average) before such packages are deprecated. Lastly, we find that our approach complements commonly used popularity metrics such as dependents, downloads, stars, and forks when detecting packages in decline.

Our work makes the following contributions:

- Propose a scalable approach to detect packages in decline using package centrality.
- Empirically evaluate our approach on the npm ecosystem.
- Create a tool prototype to facilitate the usage of our approach by practitioners.
- Make all of our dataset (i.e., the collected data, analysis results, scripts) publicly available to support replication and future research ([Mujahid, Costa, Abdalkareem, & Shihab, 2021a](#)).

The remainder of this chapter is organized as follows: We motivate our work in Section 6.2 with an example of a popular package in decline. Section 6.3 details our approach, from data collection to computing centrality trends to find packages in decline. In Section 6.4 we explain how we collect and curate the baselines we use to evaluate our approach. Section 6.5 presents the findings of our empirical study by answering our three research questions. We present a tool prototype to utilize our approach in Section 6.6. Section 6.7 describes the threats to validity. Finally, we conclude in Section 6.8.

## 6.2 Motivation Example

To illustrate the idea of using package centrality in determining a shift in community interest, we present the example of the Moment.js package. Moment.js is a JavaScript library for parsing, validating, manipulating, and formatting dates. This is a highly-used package, used in more than 1 million websites, including major companies<sup>4</sup> such as CNN, Microsoft Teams, LinkedIn and Dropbox. Moment.js was developed using a now old-fashioned JavaScript packaging method, including all its functionalities in a single bloated JavaScript class. Consequently, all websites that use Moment.js have to include the entire package regardless of the feature used, which incurs in an unnecessary overhead for website applications ([Johnson-Pint, 2020](#)).

---

<sup>4</sup>Reported by wappalyzer.com in January 2021

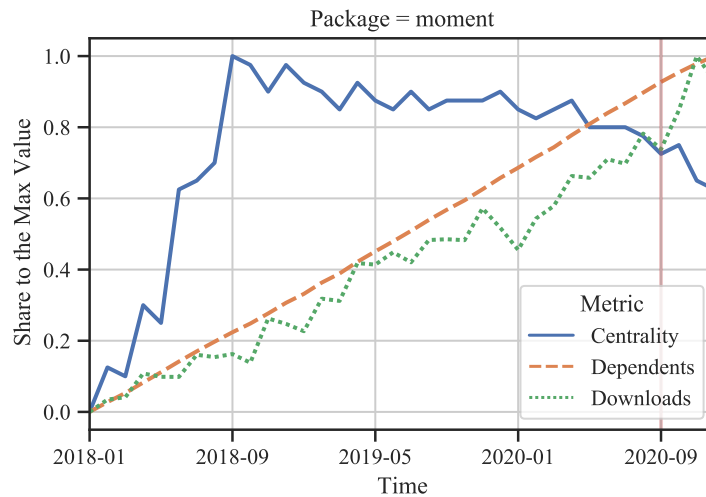


Figure 6.1: Evolution of the Moment.js package on the centrality (PageRank), the number of downloads, and the number of dependents. The red vertical line indicates the time where maintainers reported Moment.js is now a legacy project. We normalize metric values using the min-max method where values range from 0 and 1 (Codecademy, 2021).

*“Moment was built for the previous era of the JavaScript ecosystem. The modern web looks much different these days.”*

Since 2018, alternatives to Moment.js (e.g. date-fns and Day.js) have become more and more popular by providing similar functionality without incurring the overhead that Moment.js incurs. Hence, the npm community started shifting towards using more lightweight packages. This shift includes migrating well-established open source projects like Google Chrome’s Lighthouse, Vault by HashiCorp, and Web Stories by Google from Moment.js to other alternative packages (Birchler, 2020; Daley, 2018; Nanavati, 2020).

The popularity of the alternative packages led to a consistent decrease in Moment.js’s centrality in the ecosystem starting in September 2018, which can be seen clearly in Figure 6.1. On September 15th, 2020, the maintainers of Moment.js issued a statement in the README file indicating that the package is now a legacy project. While maintainers have committed to still maintain the project, they recommend that users choose a different package.

The community’s shift from using one of npm’s most used packages to other alternatives was public knowledge; however, none of the common popularity metrics, including the metrics used by the npm search engine (*npmjs*) were able to capture this phenomenon. In fact, the number of downloads of Moment.js continued to increase (as shown in Figure 6.1) as well as the number dependent packages. As of January 2021, the npm registry shows that 49,544 npm packages depend on Moment.js and it is downloaded more than 16 million times a week. The only metric that showed Moment.js’s important decrease in npm was centrality, which started to decrease as early as October 2018, the same year that alternative packages started to become more popular.

There are a couple of possible reasons why the number of downloads and dependents did not capture the decline of Moment.js. First, since thousands of projects already use Moment.js, it will continue to be downloaded every time any of these projects get installed. Even when these projects migrate to use alternative packages, it will take much longer to reflect on the number of downloads due to technical lag where developers take a long time to update their dependencies (Decan et al., 2018a). Second, as npm continues its exponential growth, newly created packages may still depend on Moment.js and substitute the core community that has migrated to the alternative solutions. Package centrality, calculated with PageRank, accounts for not only the sheer number of dependents, but the importance of dependents in the network, which aptly captures the decline of Moment.js. This example motivated us to investigate if package centrality trends can be used to identify packages that have declined in the community interest.

### **6.3 Approach**

In this section, we explain our approach that uses the trend in the package’s centrality in the npm ecosystem and detect packages in decline.

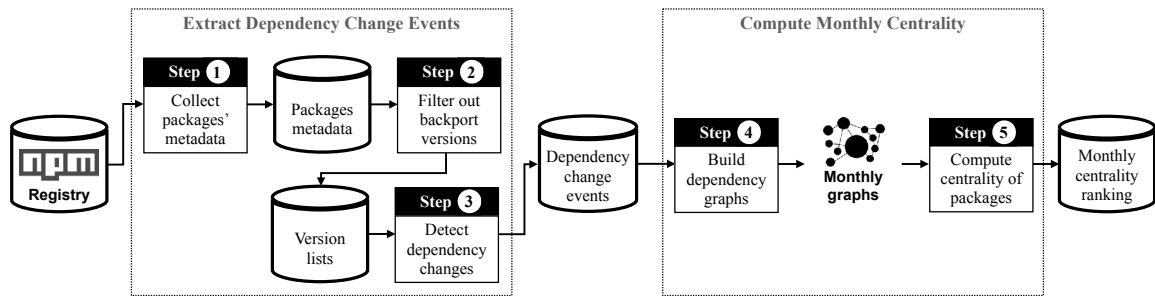


Figure 6.2: The approach to calculate centrality trends.

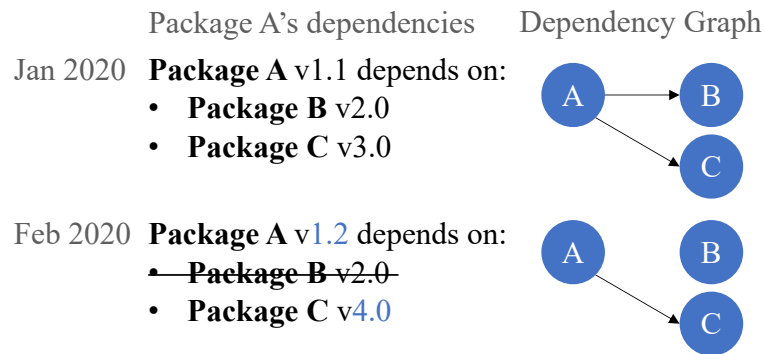


Figure 6.3: Illustration of our dependency graph build process

### 6.3.1 Calculating Centrality Trends

Since the core idea of our approach is to use centrality, we need to efficiently calculate the centrality trends of packages. We first build a dependency graph containing all packages in npm as nodes, and their dependency relationships as edges. We update this graph monthly with newly established dependencies and packages and compute the centrality metric for all packages. Each month, we rank the packages based on the value of their centrality metric. In the following, we explain the attributes of our dependency graph, then, we describe our approach, illustrated in Figure 6.2, which includes how we: i) collect and format the required metadata to build the dependency graph incrementally, and ii) build the dependency graph each month to compute the centrality metric for all packages in the npm ecosystem.



**Attributes of Our Dependency Graph.** In order to use the package centrality as an indicator for packages in decline, our dependency graph needs to have two important properties:

- (1) **Version insensitive nodes:** the nodes in our graph represent npm packages, regardless of their versions. For instance, the popular package React has 298 distinct versions released in the npm registry, but we represent it by only one node in our dependency graph. We do this because we are interested in capturing the usage shift without being affected by the technical lag in the dependency network, caused by developers taking a long time to update a dependency version (Decan et al., 2018a; Zerouali et al., 2018).
- (2) **Release sensitive edges:** an edge  $A \rightarrow B$  in our graph represents the dependency between the latest released version of package A on any version of package B. Once a new release of package A no longer depends on B, our dependency graph needs to reflect that by removing the  $A \rightarrow B$  edge. However, we do not consider backport versions as the latest released versions since they are not consistent with the package evolution time series.

To better illustrate how this dependency graph is built, Figure 6.3 presents an example of one package’s dependencies and how they are reflected in our dependency graph. As shown in Figure 6.3, the graph in each month (January and February) uses the latest version of Package A to add the edges from node A to its dependencies, but disregards the versions of the dependencies (packages B and C). Once package B is removed from A’s dependencies (in February), we remove the edge  $A \rightarrow B$  in the dependency graph. It is important to note that, by not accounting for versions in the nodes, this dependency graph is different from the dependency graph that npm resolves to install new package versions when running the `npm install` command (npm Docs, 2021).

**Extracting Dependency Change Events.** To build the npm dependency graph, we need

to extract and process all events that changed dependencies for all npm packages. In our study, we need to process two types of dependency change events for all npm packages: 1) the addition of a new package dependency and 2) the removal a package dependency. Since our dependency graph does not consider the package versions in their nodes, there is no need to account for events updating a package dependency version. We use the npm registry database to extract all the package dependency change events. The npm registry keeps a copy of the `package.json` file of all npm packages in its database for all package versions. The `package.json` file includes the list of maintainers, package description, keyword, license, the address of the source code repository, and the list of package dependencies. The registry stores each package as a document that contains its metadata.

The npm registry is powered by an Apache CouchDB database, which has a feature to set up a continuous stream of its data ([npm Documentation, 2019](#)). The feature is typically used to set up continuous replication from the registry database. We utilize this feature to retrieve a stream of all documents from the npm registry (Step ①). For each document in the stream, we filter out the irrelevant documents (e.g., design documents) and for each package we collected the `package.json` file for each of its versions.

When we build the monthly dependency graph, we only use the most recent version of each package version to create our dependency graph. Hence we order every package release by its release date. However, not all releases represent the stage of the package project at the target time. Backports are commonly employed by package maintainers to fix older releases, and they could include old dependencies that no longer appear in the package's latest releases. Hence, we filter out any release with a lower semantic version than its predecessor in relation to their respective release date (Step ②). For example, package A has released the version `3.6.0` in March 2020, but released a backport fix `2.1.0` in April 2020. Because the version `2.1.0` is smaller than version `3.6.0`, we disregard the version `2.1.0` in our analysis.

Finally, we extract the changes in the list of dependencies between versions (Step ③). We represent each change in the dependencies list as a dependency change event, which can be either an add or a remove event. When a package releases its first version, we consider each of the dependencies required by that version as an add dependency change event. In the following versions, we compare the list of dependencies on each version with the list in the previous version. If the dependency is absent in the newer version, we consider it to be a remove event; conversely, if the dependency is absent in the older version, we consider it an add event.

**Computing Monthly Centrality.** To obtain monthly snapshots of the centrality trends, we compute the centrality for the packages in the npm ecosystem each month. Consequently, we need to build a dependency graph at each month of analysis. Building separate graphs from scratch for every month can be an expensive operation and unpractical option, particularly for npm, which contains more than a million packages. To address this, we build the dependency graphs incrementally using the add and remove dependency change events that we explained previously.

In this study, we are interested in investigating the package centrality trends since the creation of the npm ecosystem. In particular, we study the period from December 2010 to December 2020. To do so, we build the first graph up to December 2010 and calculate the centrality for every package in that graph (Step ④). Then, for each month, we update the graph snapshot to reflect the monthly changes in the ecosystem. In total, we build 121 different versions of the dependency graph for the npm ecosystem, one for each month between December 2010 and December 2020.

We use the monthly dependency graphs to compute the centrality of packages in the npm ecosystem (Step ⑤). In order to compute the centrality, we use the Google PageRank algorithm (Brin & Page, 1998; Page, Brin, Motwani, & Winograd, 1999). The algorithm is commonly used to rank software artifacts, e.g., JavaScript packages (Kashcha, 2017;

Wittern et al., 2016) and Java components (Inoue, Yokomori, Yamamoto, Matsushita, & Kusumoto, 2005). The PageRank algorithm is a variant of the Eigenvector Centrality metric, which measures the importance of each node within the graph based on the number of incoming edges and the importance of the corresponding source nodes. The underlying assumption of PageRank is that a node is only as important as the nodes that link to it (Gleich, 2015; Manaskasemsak & Rungsawang, 2005). In our study and through the use of PageRank, the package centrality score is affected by both the number of dependent packages and the score of the dependent packages themselves. Thus, packages obtain higher scores if their dependent packages themselves have high scores.

However, the centrality value of nodes in PageRank decays over time as the network grows (Berberich, Bedathur, Weikum, & Vazirgiannis, 2007). This may impact the evolution analysis and means it is not meaningful to compare centrality values of packages on different periods as these will always tend to decrease (at least for growing networks). To address this, we focus instead on analyzing the ranking of the nodes' centrality. Once we compute the centrality for all packages on a particular month, we rank the packages based on their centrality values  $(v_1, v_2, \dots, v_n)$  where  $v_1$  is the most central package and  $v_n$  is the least central package similar to prior work (Wittern et al., 2016). Finally, we invert the ranking in negative values  $(-1 \times n)$  to give a higher ranking value to the more central packages, and make the centrality ranking comparable to other metrics (e.g. downloads), where a higher value means higher importance. With this, we have the centrality ranking position evolution for each package in the npm ecosystem since its creation up until December 2020.

### 6.3.2 Detecting Packages In Decline

Now that we have the evolution of all packages' centrality rankings, we use it to provide a reliable method to identify packages in decline. To classify a package as in decline, we

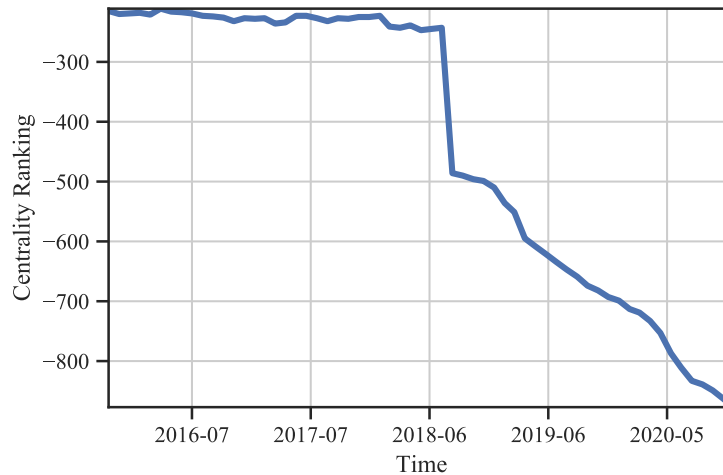


Figure 6.4: Example of a package trend in decline.

use its centrality trend of the latest six months. We fit a linear function using the least-squares regression ([NIST/SEMATECH, 2012](#)), then we analyze the slope ( $m$ ) of the trend to identify a package in decline. In our study, a package is classified in decline if its centrality trend shows a significant negative slope:

- (1) **Slope:** the slope of the centrality trend for the last six months should be  $m < v$ , with default  $v = 0$ .
- (2) **P-Value:** to test whether the negative slope is statistically significant, we perform the Wald Test with a conservative p-value ( $p$ ) threshold, i.e.,  $p < \alpha$ , with default  $\alpha = 0.001$  ([Judge, Griffiths, Hill, Lutkepohl, & Lee, 1985](#)). The Wald Test is a way of testing the significance of particular explanatory variables in a statistical model.

In practice, our approach classifies packages as in decline when they have consistently fallen down in the npm centrality rankings for six months. Figure 6.4 shows an example of the package `istanbul-api`, which is classified as in decline, with a clear decrease in the centrality rankings starting from mid 2018. This decline can be justified by the incompatibility of the package with new Javascript features ([Farrell, 2019b](#)), which led to the deprecation of the package later in April 2019 ([Farrell, 2019a](#)).

## 6.4 Evaluation Datasets

To obtain a baseline for our approach, we devise a dataset containing packages in decline and packages not in decline, so we can evaluate if our approach can reliably report packages in decline. Unfortunately, there is no existing large dataset that captures the shifts in community interest we aim to evaluate.

To compensate for the absence of this ideal dataset, we build three different baseline datasets. First, we build a corpus using metrics from the official search engine of npm (*npms*) to evaluate if centrality can detect packages in decline before *npms* (Section 6.4.1). Second, we collect data from the largest survey of the JavaScript community conducted by [Benitte and Greif \(2021\)](#), which asked the opinion of more than 20 thousand developers about 20-30 popular npm packages (Section 6.4.2). With this baseline we aim to evaluate if centrality can capture the satisfaction/dissatisfaction of developers using the trend in centrality right before the survey took place. Third, we craft a dataset of deprecated packages to evaluate if our approach can help identify the decline in popularity well before the maintainers deprecated the packages (Section 6.4.3).

### 6.4.1 Extracting *npms* Validation Baseline Corpora

One of the most reliable platforms developers use to select npm packages for their projects is the official npm search engine, *npms* ([Abdellatif et al., 2020](#); [npms, 2016](#)). The *npms* engine continuously analyzes the npm ecosystem, and collects 27 package metrics from different sources (e.g. package repositories on GitHub). Using the collected metrics, a final score for each package is calculated based on three different aspects i.e., quality, popularity, and maintenance ([Abdellatif et al., 2020](#); [npms, 2016](#)). The higher the score of a package, the more popular, better quality and better maintained the community perceives the package to be. Hence, a steep decline in a package score can be used as an indicator of a package in decline and a stable or increasing score can be an indicator of a package not

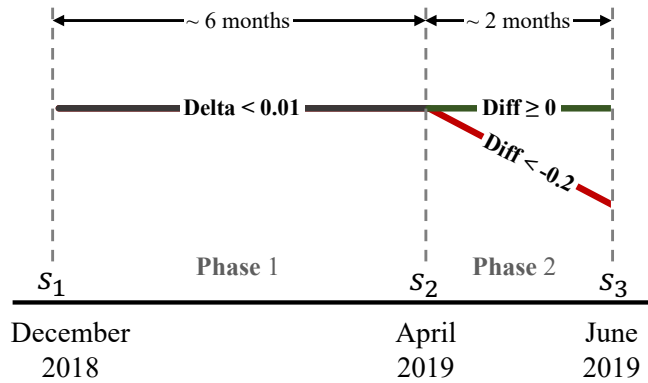


Figure 6.5: Timeline used to select validation baseline from *npm*s.

in decline.

It is important to note however that we want to evaluate the hypothesis that *centrality can better identify shifts in community interest than currently used metrics*. Hence, we want to craft a dataset that allows us to use *npm*s score as validation, but is not directly influenced by the *npm*s score. To this aim, we craft the dataset using a multi-phase approach, as illustrated in Figure 6.5. We first select packages that have shown a stable *npm*s score during a period (Phase 1), and use this same period to evaluate the centrality of a package. Because the score metric is stable during this period, one would not be able to classify the packages in decline from not in decline just by analyzing *npm*s score, and we can be sure centrality is not influenced by already reported metrics. Then, in the subsequent period (Phase 2), we label packages in decline as the ones that have experienced a sharp decline in the *npm*s score and label packages not in decline as the ones that have either remained stable or increased their *npm*s score. Using this process to craft the baseline, we also have a starting date for packages in decline given by the *npm*s score, which is at the earliest the start of Period 2. We can use this point in time, to evaluate how much in advance (if any) our approach can detect that a package is in decline before the decline is shown in the *npm*s score.

One limitation of using the *npm*s metrics is that *npm*s does not store the historical values

of its packages' score. We cannot pick any period interval for Phase 1 and Phase 2 and are limited by the snapshots of the entire *npm*s score ranking we collected in the past. We collected *npm*s packages' scores on December 2018, April 2019, and June 2019 and we use the period of December 2018 to April 2019 as Phase 1, and use the period of April 2019 to June 2019 as Phase 2 of our dataset baseline.

All *npm*s scores vary from 0 (very low) to 1 (perfect score). We start crafting our dataset by selecting packages that have a score 0.7 or higher, to prevent our analysis from focusing on very low-quality packages. As we showed in Figure 6.5, in Phase 1 we consider all packages that have a score variation smaller than 0.01, which indicates to be relatively stable. Then, we label as packages in decline, all packages that have exhibited a negative change in the *npm*s score between  $S_2$  and  $S_3$  by more than 0.2 score points. We label as packages not in decline, all packages that have exhibited the same score or higher between  $S_2$  and  $S_3$ . At the end of this process, this dataset contains a total of 4,457 packages, with 2,259 being labeled as in decline and 2,198 labeled as not in decline.

The three thresholds used in the above methodology were determined as follows: the first threshold of 0.01 is the tolerance in the *npm*s score deltas in Phase 1. This threshold equals the mean value of changes in the *npm*s score between  $S_2$  and  $S_3$  and it is small enough to guarantee that package scores are stable for at least 6 months before April 2019. The second threshold 0.2 is the minimum decrease in the *npm*s score to label a package as in decline. This threshold is equal to the value of standard deviation over the *npm*s scores and it is large enough to capture the significant score changes. The last threshold, 0.7, is the minimum *npm*s score for a package on  $S_3$  to be considered in our baseline dataset. This will minimize the risk of mislabeling our baseline by including low-quality packages with very low *npm*s scores and it is a good compromise between the dataset size and quality.



## 6.4.2 Survey Validation Baseline Corpus

We want to evaluate if our approach can capture the shifts on the interest and satisfaction of the npm community with popular packages. While we cannot craft a dataset that reliably captures the npm community interest without surveying a very large sample of JavaScript developers, we opted to use the data from the largest survey available on the JavaScript ecosystem: the State of JavaScript survey ([Benitte & Greif, 2021](#)).

The State of JavaScript survey is an extensive survey conducted by [Benitte and Greif \(2021\)](#) to assess the JavaScript community's views. In 2019, the survey had a total of 21,717 respondents all across the globe ([Greif & Benitte, 2019](#)). The survey's primary focus is to ask JavaScript developers their opinion on a set of popular npm packages. Then, the survey ranks each package according to four categories:

- (1) **Awareness:** share of total respondents that reported to have heard about the package. This category includes both developers who have experience using the package and developers never use the package before.
- (2) **Usage:** share of total respondents that have used the package in their projects. This category does not consider if the developer is satisfied with using the package.
- (3) **Interest:** share of respondents who did not use the package but are interested in using it in the future.
- (4) **Satisfaction:** share of respondents that have used the package in the past and will continue to use it.

To use the survey results, we use its GraphQL API<sup>5</sup> to retrieve the summary of the responses for each package.

---

<sup>5</sup><https://graphiql.stateofjs.com>

### 6.4.3 Deprecated Packages Corpus

With this third corpus, we want to evaluate if our approach can help identify packages in decline that have eventually been deprecated by maintainers. Deprecated packages should not be reused by other packages or JavaScript applications and npm warns developers when they install deprecated packages. The goal of our analysis is to evaluate if centrality trends can capture the decline in the community interest well before the package is flagged as deprecated, which can help developers to migrate from these packages while they are still being maintained.

To craft this dataset, we need to collect a list of deprecated packages from the npm ecosystem. Similar to Section 6.3, we started by retrieving the metadata for all packages from the npm registry. Then we capture metadata for packages with a deprecation message, which left us with a list of 44,857 packages. However, developers use the npm deprecation feature for various reasons, including renaming or merging packages. The following quote is an example of a deprecation message for a package whose maintainers used the deprecation feature to change the package name.

*“Jade has been renamed to pug, please install the latest version of pug instead of jade (Lindesay, 2016).”*

To create a valid list of deprecated packages, we select the top 1,000 deprecated packages based on their *npm*s score on June 16th, 2019. Then we manually classify packages to filter out cases where they are not an actual deprecation. For this aim, first, we verify if the deprecation note discloses clearly that a package is actually deprecated. If the deprecation message is not clear, we check the project status from the package’s readme file, then the repository’s readme file. If needed, we follow relevant links in the deprecation messages or the readme files to remove ambiguity. Finally, if the deprecation message mentions another package’s name, we check if both are pointing to the same repository; if so, we examine the repository and its history to classify the case as a rename or not. After applying our

manual classification process, we find that only 556 out of the 1,000 packages are actual package deprecation cases. We use these 556 packages in our analysis later in the study.

## 6.5 Results

This section describes our research questions. For each research question, we explain its motivation, illustrate our approach to answer the question, and discuss the findings.

### **RQ1: How effective is our approach in detecting packages that are in decline?**

**Motivation.** In this question, we investigate the performance of our approach of using the centrality trend to identify packages in decline. The decline of package centrality could be a symptom that better alternatives have emerged or a shift happened in the community interest. In the scientific literature, centrality has been used in many disciplines such as social networks to identify the central node of a network (e.g., (Cadini et al., 2009; Hong et al., 2014; Maharani et al., 2014; Stergiopoulos et al., 2015)) and software engineering to understand the significance of software components (e.g., (Inoue et al., 2005; Wittern et al., 2016)). If the approach can aptly capture packages in decline, it can be embedded in package search engines, such as *npm*s, to increase developers' awareness of the community interest and help them make a better-informed decision to select or reevaluate their package dependencies.

**Approach.** We craft a baseline as described in Section 6.4.1 to evaluate our approach as a binary classification problem. Then we use our approach to classify packages into two classes: in decline and not in decline.

As mentioned in Section 6.4.1, packages labeled in decline are packages that have experienced a sharp decline in the *npm*s ranking in a short period of two months, i.e., between  $S_2$  and  $S_3$ . We calculate the centrality in the last six months before  $S_2$ , when the packages were still stable in the *npm*s rankings. This ensures that we evaluate if the centrality can be used as an early detector of packages in decline that only later will be observed in the *npm*s rankings. Then, as described in Section 6.3.2, we classify packages that have a negative centrality trend slope (i.e.,  $m < v$  with default  $v = 0$ ) as in decline and other packages as not in decline.

To evaluate the performance of our approach in identifying packages in decline, we report the well-know performance measures: precision ( $P$ ), recall ( $R$ ), and  $F_1$  score. In the context of our evaluation, precision is the percentage of packages classified as in decline that are actually in decline (i.e.,  $Precision = \frac{T_p}{T_p + F_p}$ ), where  $T_p$  is the number of packages labeled as in decline that are correctly classified as in decline;  $F_p$  denotes the number of not in decline packages classified as in decline. Recall is the percentage of packages that correctly classified as in decline relative to all of the packages that are labeled as in decline (i.e.,  $Recall = \frac{T_p}{T_p + F_n}$ ), where  $F_n$  measure the number of packages in decline that classified as not in decline. We then combine both precision and recall using the well-known  $F_1$  score (i.e.,  $F_1 = 2 \times \frac{P \times R}{P + R}$ ).

In addition, to mitigate the limitation of choosing a fixed slope threshold (i.e.,  $v = 0$ ) when calculating precision and recall, we also present the Area Under the Receiver Operating Characteristic Curve (ROC-AUC) value. ROC-AUC is computed by measuring the area under the curve that plots the  $T_p$  rate against the  $F_p$  rate while varying the slope threshold used to determine if the approach should classify a package as in decline or not. The ROC-AUC's main merit is that it reports the performance independently from the used threshold; it is also robust toward imbalanced data since its value is obtained by varying the classification threshold over all possible values (Lessmann et al., 2008; Nam & Kim,

Table 6.1: Results of using the centrality trend to classify packages from the *npm*s validation baseline.

Dataset	Total cases	4,457
	In decline	2,259
	Not in decline	2,198
Performance	True Positive ( $T_p$ )	1,969
	False Positive ( $F_p$ )	498
	Precision ( $P$ )	0.80
	Recall ( $R$ )	0.87
	$F_1$ score	0.83
	ROC-AUC	0.90

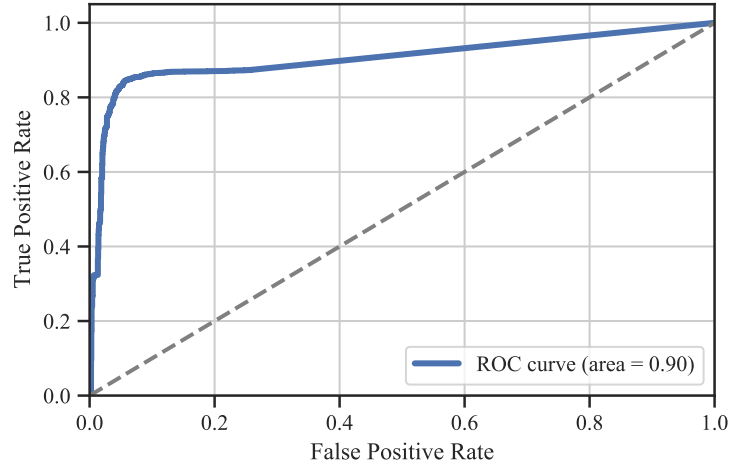


Figure 6.6: ROC curve with the AUC value for the evaluation based on the *npm*s baseline.

2015). The ROC-AUC has a value that ranges between 0 and 1, where a higher ROC-AUC value indicates better classification performance.

**Results.** As shown in Table 6.1, we evaluate our approach on 4,457 npm packages where 2,259 are labeled as in decline and 2,198 are labeled as not in decline. The results show that our approach of using the centrality trends correctly identifies 87% of the packages in decline with a precision equal to 0.80. That is, for every five packages classified as in decline, four were correctly classified and one was wrongly flagged as in decline. This indicates that our approach can aptly identify packages in decline before they are actually shown in the *npm*s rankings, with an  $F_1$  score of 0.83 and ROC-AUC of 0.90 As shown in

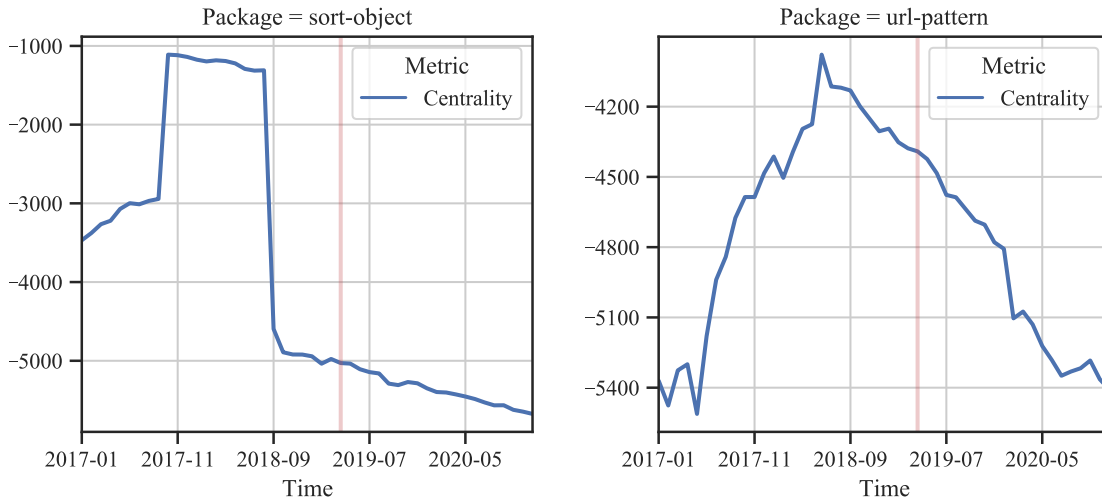


Figure 6.7: Examples of packages that our approach only detected the decline after the *npm*s score. The red vertical line indicates the time of  $S_2$ .

Figure 6.6. The figure shows the false positive rate on the x-axis and the true positive rate on the y-axis, while the solid line represents the value of each of them based on a range of possible thresholds.

We analyze the 290 packages that were in decline, but where our approach could not identify their decline using centrality. Out of the 290 cases, 217 (74.83%) packages exhibited a centrality decrease only after April 2019 ( $S_2$ ), showing that in these cases the *npm*s metrics decrease before the centrality. Figure 6.7 shows examples of packages that our approach could not detect packages in decline in advance of *npm*s. In the figure, both packages show a decrease in centrality before April 2019 ( $S_2$ ). However, our approach requires a statistically significant decrease over a six months period, with a very conservative default threshold  $\alpha = 0.001$  to detect the packages as in decline. Hence, our approach detected the packages as in decline after  $S_2$ , when the decline became statistically significant.

We also examine the 498 packages that were not in decline, but where our approach wrongly identifies them as in decline. We observe that out of the 498 cases, 384 (77.11%) packages have less than 100 dependent packages. In the rest of the false positive cases,

we observe that most of the packages (112 out of 114) have their number of dependents increasing, however their centrality is decreasing. For example, the package `mongoose` is a popular object modeling tool whose dependents increased from 4,995 to 5,568, whereas its centrality ranking declined from -443 to -484. The main factors behind these false positive cases can be explained by the following:

- (1) The dynamic of the centrality ranking tends to punish packages that do not gain more dependents (directly or indirectly) on them compared to other packages in the same ranking tier. In the `mongoose` example, even with the 11.47% increase in the number of dependents, the number of dependents and their centrality were not enough to maintain the centrality ranking compared to other packages in the same ranking tier.
- (2) In packages with a small number of dependents, the centrality trend can be affected by a small number of community members that do not reflect the overall community interest. This could explain 52 (17.93%) of the 498 false negative cases and 384 (77.11%) of the 498 false positive cases.

**Impact of Moving Averages.** Simple moving averages (SMA) is a technique used to reduce the noise in the time-series data (James, 1968). In this RQ, we use the trend of the monthly centrality rankings to detect packages in decline. However, using the SMA to smoothen the trends may result in improving the performance of our approach. In our context, we experiment using the technique to reduce the effect of noise in the monthly centrality data. To do so, we re-run our experiments on the *npm*s validation baseline. For each package in the baseline, we compute the simple moving averages (based on 4 months average) for its monthly centrality rankings. Next, we apply our approach in detecting the centrality decline on the SMA values. The result of the experiment shows that incorporating the moving averages improved the precision of our approach from 0.80 to 0.85. However, it slightly decreases our approach's recall from 0.87 to 0.83, while keeping the F1-score

almost constant (from 0.83 to 0.84). Finally, since using the moving averages requires more extended history, the number of packages that our approach can be applied on is reduced slightly from 4,457 to 4,272 packages.

The result shows that our approach can correctly detect **87%** of packages in decline with a precision of 0.80, an F1-score of 0.83 and an ROC-AUC of 0.90.

## **RQ2: How early can our approach detect packages that are in decline?**

**Motivation.** Once we learned that our approach is effective in identifying packages in decline, we would like to know how early in advance can our approach detect the decline. Identifying packages in decline as early as possible is essential for taking proactive action to mitigate the decline of the package. Also, it increases the awareness of the community about possible better alternatives by allowing developers to avoid selecting declining packages and to pay more attention to the alternatives that are increasing in centrality. Package maintainers can also use our approach as a sign of a decrease in community interest in their package, which can motivate them to remediate possible causes of dissatisfaction or make them focus on other solutions altogether. Furthermore, developers that reuse packages can use the centrality trend as an early indicator of decline to look for alternatives long before their dependencies become unmaintained.

**Approach.** To evaluate how early our approach can detect packages in decline, we employ a sliding window technique. Since we calculate centrality at the granularity of months, we slide the analysis window back in time, sliding our window of six months one month at a time. We recalculate the in decline analysis after each window sliding (i.e., month) by applying the same method explained in Section 6.3.2. We continue this process as long as the in decline analysis continues to identify the package as in decline.



Note that since we use a 6-month window to detect the decline, when we report that our approach captured a package in decline 4 months in advance, this means that the slope of the centrality trend consistently decreased in the 6 months prior to these 4 months. That is, the package is exhibiting a decrease in the centrality rankings for up to 10 months.

We used our three different dataset baselines to evaluate how early our approach can detect packages in decline. We evaluate how early our approach can detect packages in decline based on all packages that our approach classifies as packages in decline.

- (1) ***npms* dataset:** This dataset was crafted from the *npms* rankings, as explained in Section 6.4.1. In this dataset, we start measuring the packages in decline before April 18th, 2019, where the *npms* score was still stable.
- (2) **Deprecated dataset:** This dataset was crafted from the deprecated npm packages, as explained in Section 6.4.3. In this evaluation, we aim to assess how far in advance we can use our approach to identify packages that have become deprecated. In this evaluation, we use the deprecation date of each package as the starting point to measure whether the package is in decline.
- (3) **State of JavaScript dataset:** We collect this dataset from the State of JavaScript survey of 2019 (Greif & Benitte, 2019), as explained in Section 6.4.2. We label packages with a share of satisfaction less than 50% as in decline and the rest of the packages as not in decline. In this evaluation, we measure the packages in decline before November 25th, 2019, which is the date of receiving the first survey response.

**Results.** Table 6.2 presents the results of our experiment, showing how far in advance our approach can detect packages in decline. The first row in the table shows that our approach classifies 2,467 packages from the *npms* dataset as in decline with an average of 18.35 (median = 12.57) months before April 2019 ( $S_2$ ). To reiterate, only after the  $S_2$  date, these packages have shown a steep decline in the *npms* scores. Our results show that half of the

Table 6.2: Results of three datasets on how early in months our approach can detect packages in decline.

Dataset	Labeled as in decline	Classified as in decline	Time (months)	
			Mean	Median
<i>npm</i> s	2259	2467	18.35	12.57
Deprecated	552	446	16.15	13.29
Survey	4	3	13.13	4.80

packages were experiencing consistent centrality decline for more than a year before this decline was captured by the *npm*s metrics.

The second row in Table 6.2 shows the results for the deprecated dataset. Our approach was able to identify the centrality decline on average more than a year (16.15 months) before the packages became deprecated. Also, the decrease in the centrality rankings captures the decline of 446 out of 552 deprecated packages. Our results indicate that the centrality trend can be used as an early indicator of deprecated packages, with a good recall, capturing 80% of the deprecated packages.

Finally, the third row in Table 6.2 shows the results of our evaluation using the State of JavaScript survey dataset. Our approach correctly classified three out of the four labeled in decline packages with an average of 13.13 (median = 4.80) months before the first survey response date without any false alarms.

Figure 6.8 shows the distribution of time in months for how early our approach can detect packages in decline across the three datasets. The figure shows that our approach detects 25% of packages in decline more than 31 months before the significant *npm*s score decrease, and 22 months before a package got deprecated.

The results show that our approach can detect packages in decline on average **18.35 months** before the *npm*s score declines. Also, it detects packages in decline on average **16.15 months** before a package gets deprecated.

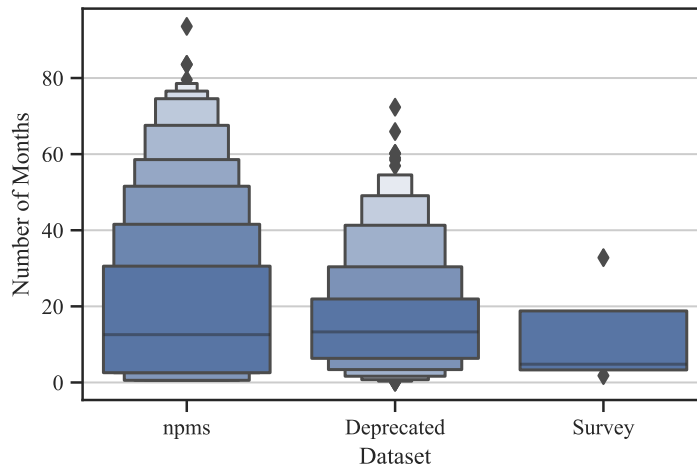


Figure 6.8: Letter-value plots for the distribution of how early our approach can detect packages in decline.

### RQ3: How does our approach compare to other metrics in detecting packages that are in decline?

**Motivation.** After determining our approach’s effectiveness in detecting packages in decline, months in advance, we would like to know if other widely used metrics already capture (or complement) the information centrality indicates. There are already several metrics, e.g., as the number of GitHub stars from their repository project, that aim to provide a popularity indicator of npm packages and have been used by prior work, (e.g., (Abdellatif et al., 2020; Borges et al., 2016; Borges & Valente, 2018; Papamichail, Diamantopoulos, & Symeonidis, 2016; Zerouali et al., 2019; Zhu et al., 2014)). If centrality is already properly captured by other widely used metrics, there is no incentive to incorporate centrality in the current package platforms. If the centrality trends, however, provide a new perspective on the popularity and community interest of a package, there is a good motivation to make the centrality information more accessible to developers to improve their community awareness.

**Approach.** We are particularly interested in assessing how much of the centrality is already captured by metrics that the *npms* analyzer uses. In particular, we studied metrics that

present the number of dependents, number of package downloads, Github stars, and Github forks. We evaluate if the centrality trend correlates with these metrics and whether we could use these previously used metrics to detect packages in decline, with similar or better performance than our centrality trends.

To compare our approach with the other metrics, we start by collecting the monthly number of dependents, downloads, stars, and forks of 40,619 packages in npm. We retrieve the number of monthly dependents using the dependency graphs we build to measure the centrality, explained in Section 6.3.1. For the number of downloads, we use the npm REST API<sup>6</sup> to collect the daily number of downloads for the time between each package creation date (not before February 2015, which earliest data that the API keeps) until December 2020. Then we aggregate the daily downloads for every month.

The GitHub API does not provide an endpoint to retrieve the historical number of stars and forks. To overcome this challenge, we rely on the API of Porter.io,<sup>7</sup> a service that analyzes Github continuously and retrieves the historical number of stars and forks for a wide range of repositories. Thus, we use Porter.io's API to collect the historical number of Github stars and forks for package repositories with more than 100 stars in *npmjs* at December 27th, 2020. We omit packages with fewer than 100 stars, to prevent our analysis from being dominated by packages that are seldom used by the community.

After collecting the metrics for all packages with more than 100 stars, we notice that not all packages have sufficient data for our analysis. For instance, some packages lack sufficient historical data or one or more of their metrics have all the data points as zero, e.g., packages that have no dependents. Therefore, to simplify our analysis and report results from a uniform dataset, we exclude packages that do not have sufficient information for all metrics. This step excluded 21,201 packages from the initial set of 40,619; thus, our analysis is based on 19,418 packages.

---

<sup>6</sup><https://api.npmjs.org>

<sup>7</sup><https://porter.io>

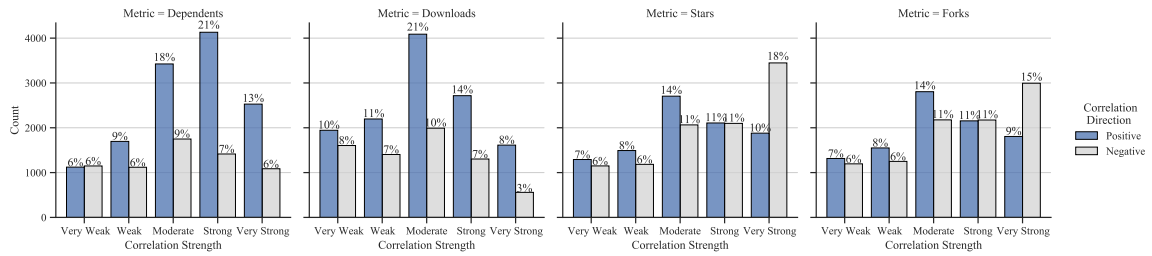


Figure 6.9: The distribution of the correlation between centrality and the metrics.

To evaluate if the other metrics' trends indicate the same trend as centrality rankings, we test the correlation between the monthly centrality trend and each of the other metrics' monthly trends. We use Spearman's rank correlation test, and we apply the correlation test on the metrics for each package separately. We use Spearman's rank correlation coefficient since our dataset is not normally distributed (Kendall, 1938). Spearman's rank correlation coefficient ( $\rho$ ) has a value that ranges between +1 and -1. In our context, +1 means that a metric value always increases when the centrality increases and -1 means that a metric value always decreases when the centrality increases. A Spearman ( $\rho$ ) of zero indicates no correlation between the metric and the centrality (Fowler, Cohen, & Jarvis, 2009; Kendall, 1938).

**Results.** Figure 6.9 shows the distribution of Spearman's rank correlation coefficient ( $\rho$ ) between the centrality trend and the trend of each of the other popular metrics. Following the guidelines of Fowler et al. (2009), we group the correlation distribution into five intervals: very weak correlation (0.00 to 0.19), weak correlation (0.20 to 0.39), moderate correlation (0.40 to 0.69), strong correlation (0.70 to 0.89) and very strong correlation (0.90 to 1.00). The figure plots the correlation results for 19,418 packages. We observe that centrality and the evaluated metrics have correlations that spread all the spectrum from a perfect positive correlation ( $\rho = +1$ ) to a perfect negative correlation ( $\rho = -1$ ). Overall, this shows that centrality is not aligned to the other metrics for most packages, indicating that centrality may provide new information that is not captured by the other metrics. Next,

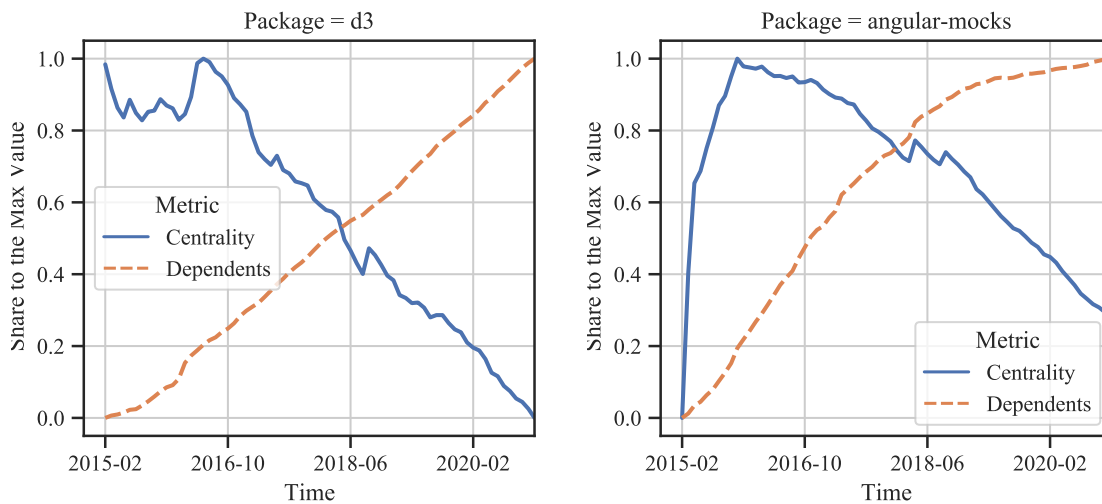


Figure 6.10: Examples of packages with strong negative correlation between the centrality trend and the number of dependents trend. We normalize metric values to range between 0 and 1.

we discuss the comparison to each metric and its implications.

As shown in Figure 6.9, the dependents metric shows the strongest correlation with centrality amongst the evaluated metrics. Roughly a third of the packages (34%) have a strong or very strong correlation between its number of dependents and centrality. This is somehow expected since packages with high centrality tend to have many dependents and vice versa. Still, this strong correlation does not hold for the majority of packages that we evaluated because our approach to calculate the centrality uses an algorithm that considers not only the number of dependents but also the importance of each of them. This explains why 13% of the packages have a strong negative correlation between the number of dependents and centrality. Such packages, such as the examples in Figure 6.10, have shown a steady increase in the number of dependents but an equally steady decrease in their centrality in npm.

The number of downloads also has a strong positive correlation with centrality in 22%

of the packages. Similar to the case of the number of dependents, it is expected that packages that rise in the centrality ranking will have an increase in the number of downloads. In 36% of the cases, however, the centrality and the number of downloads are only weakly correlated (positively and negatively), and in 10% of the packages, they have shown a strong negative correlation. As shown in the Moment.js example (Section 6.2), these are the packages that, albeit having a constant increase in their downloads, are falling in the ranking and becoming less central in the npm network. These are the packages in which centrality can work best as an indicator of community interest. The number of downloads depends on the number of installed systems, which may take a longer time to reflect the package's actual community interest.

The stars and forks metrics have approximately half the packages positively correlated with centrality and half the packages negatively correlated with centrality. This is a consequence of the monotonic characteristic of stars and forks. Projects tend to always increase their number of stars/forks, as contributors only rarely remove stars from a project. In fact, in our dataset only 2.39% of the packages showed a substantial decrease in the number of stars and forks in their life cycle. Centrality, on the other hand, may increase and decrease as the community shifts its interest to the package or away from it.

To gain a better understanding of how these metrics are different, we use the following process to select four package examples: 1) We use the State of JavaScript 2019 survey that we explain in Section 6.4.2 to select popular packages. 2) The survey includes 28 npm packages; we order them based on the community satisfaction score and select a package from each quartile, i.e., Sails.js, Jasmine, React, and Jest with satisfaction scores 26%, 67%, 89%, and 96%, respectively. Figure 6.11 plots each of the four packages with their monthly trend for all metrics.

With the decrease in maintenance activities and the increase in the number of unfixed bugs, developers start discussing the quality and health of the package Sails.js ([Hacker](#)

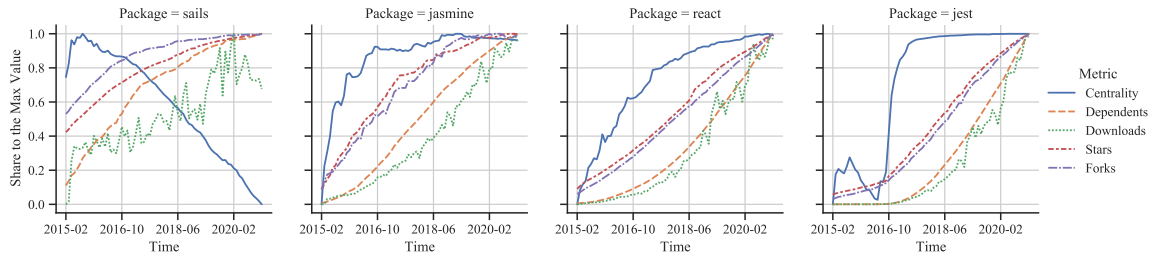


Figure 6.11: Line plots showing the trend of centrality alongside with the trend of other metrics. We normalize metric values using the min-max method where values range from 0 and 1 (Codecademy, 2021).

News, 2015a, 2015b). We observe from Figure 6.11 that the package Sails.js has a decreasing centrality trend since 2015; however, all other metrics continued to increase. The centrality trend is more consistent with the survey results, where 74% of the developers (1,166 developers) that said they used the package Sails.js responded that they would not use the package again. Even though the package decreases in centrality, the package is still increasing in the number of downloads and other metrics.

Conversely, the packages Jasmine and React, which have relatively higher satisfaction scores, show a consistent increase in the centrality trend. The package Jest showed an interesting change in the centrality evolution. The package had known performance issues until mid 2016 (Jest Blog, 2016b), where the centrality decreased. After the maintainer of Jest performed a complete rewrite of the package to overcome its issues (Jest Blog, 2016a), and having these changes well-received by the community (Hacker News, 2016), Jest started showing a significant increase in centrality. By looking at Figure 6.11, we see that only the centrality measure captures the changes of the community’s interest toward Jest.

Centrality tends to provide trends that are different from those provided by other metrics such as dependents, downloads, stars, and forks.



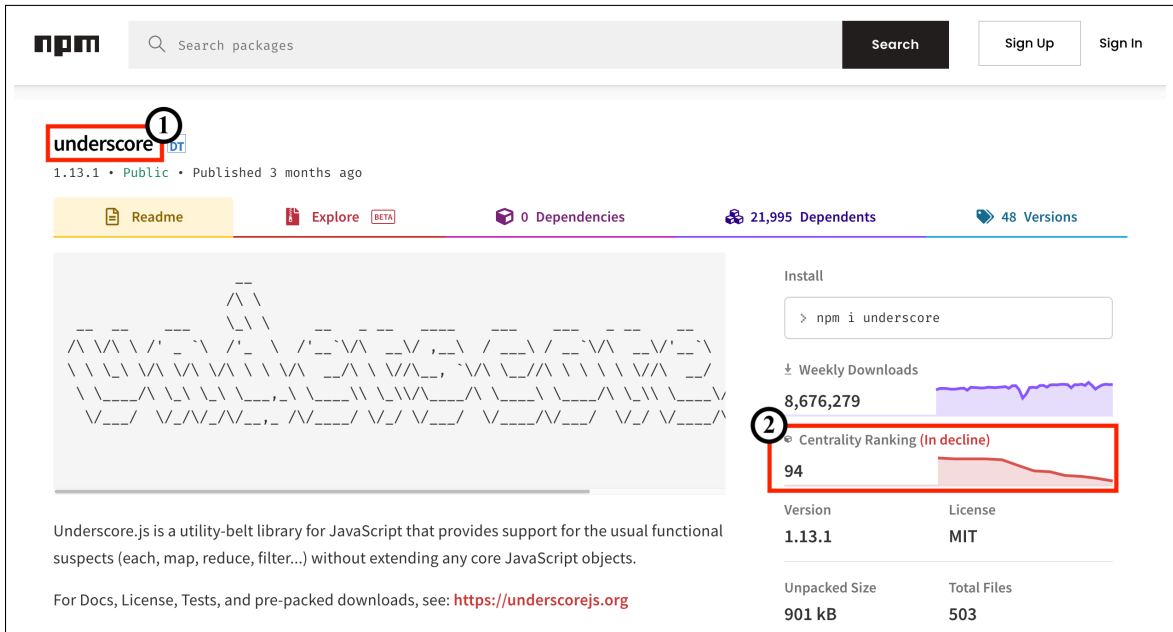


Figure 6.12: A screenshot of the npm website showing the package underscore with the integrated centrality information from our Chrome extension.

## 6.6 Tool Prototype

The main implication of our study is that reporting the centrality trends of packages as a popularity metric in npm can be very informative for developers. Developers should use the centrality trend, together with other popularity metrics, to have a better informed assessment on which packages to select. To enable this, we build a prototype web browser extension called *Centrality Checker* that uses our approach of detecting package in decline. Our prototype extension helps inform developers about the centrality trend when they browse a package on the official npm website.<sup>8</sup>

We build the tool as a Chrome Extension. Users can activate our extension in their Chrome browser. Once they browse a package on the *npm* website, our extension includes the package centrality trends and the result of examining if the package is in decline into the npm website. The initial view when a user browses a package on npm shows the centrality trend of the last year. Users can hover over the centrality trend chart to explore

<sup>8</sup><https://www.npmjs.com/>

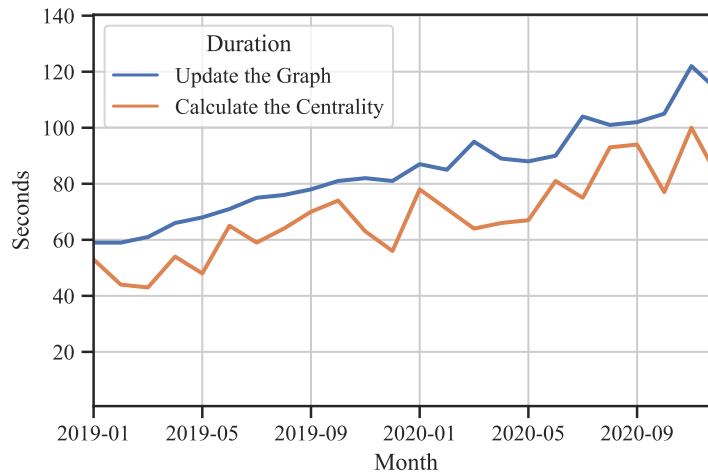


Figure 6.13: The time required to update the dependency graph and calculate the centrality for all packages. The experiment was performed on a conventional machine with an Intel Core i5 processor and 16GB of memory.

the monthly centrality ranking values from the last year. Figure 6.12 shows an example of an npm package with the proposed Chrome extension enabled. In this example, we show the package underscore ① with the centrality information embedded ②.

When a user browses a package on the npm website, the extension sends a request to a backend server to retrieve the needed data to render and embed the centrality ranking into the npm website. The backend continuously retrieves the dependency change events from the npm registry and calculate the centrality once every month as described in Section 6.3.1. The backend then determines whether each package is in decline using the approach described in Section 6.3.2. Finally, the backend caches the results to be served efficiently to our web browser extension. The tool is publicly available and can be installed through the Chrome Web Store.<sup>9</sup> Also, we open sourced the tool on Github.<sup>10</sup>

**Scalability.** With the exponential growth in the number of packages in the npm ecosystem (Decan et al., 2019), the time required to incrementally build the monthly dependency graph and calculate the centrality for all packages increases over time. In particular, as

<sup>9</sup><https://chrome.google.com/webstore/detail/centrality-checker/bmpafkghbmojppjoeienibieljacdoaj>

<sup>10</sup><https://github.com/centrality-checker/chrome-extension>

shown in Figure 6.13, the time required to update the dependency graph increased from 1 minute in January 2019 to 2 minutes in December 2020. The same goes for the time needed for calculating the centrality and detecting packages that are in decline, which increased from 50 seconds to 100 seconds. However, even with this increase, the cost of running our approach is relatively low and it can scale to handle the rapid growth of the npm ecosystem.

## **6.7 Threats to Validity**

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

### **6.7.1 Threats to Internal Validity**

Threats to internal validity are related to experimenter bias and errors. A limitation of our approach is that it only considers dependencies between packages in npm. This limitation will impact the centrality of packages that are not meant to be reused by other packages, but other JavaScript applications. Future work should investigate how to incorporate JavaScript applications in the network and how to attribute their importance in the npm network (e.g., using the number of stars in GitHub). In our approach, the package importance is calculated by the centrality of its dependents, however, applications are not meant to be reused by other projects.

Another important threat to internal validity concerns the datasets that we used as baselines when evaluating our approach. In our baseline datasets, we used various thresholds that impact which packages to include and their labeling. Since having a gold standard for npm’s community interest is very difficult, we combine evaluations made from three datasets to mitigate for the lack of a large-scale ground truth. Still, there is a need for a long term evaluation of the centrality as a complementary metric for current popularity

metrics. Future work could investigate if developers find centrality a useful metric when selecting packages. Finally, our approach may contain bugs that may have affected our results. We made our scripts and dataset publicly available to be fully transparent and have the community help verify (and enhance) our approach ([Mujahid, Costa, Abdalkareem, & Shihab, 2021a](#)).

### **6.7.2 Threats to External Validity**

Threats to external validity are related to the generalizability of our findings. Our investigation focused entirely on the npm ecosystem, which has very particular characteristics: a centralized package registry, hundreds of thousands of software packages, and a very active and popular programming language. Also, the size of packages in the npm ecosystem is relatively small compared to the size of modules and software components in other ecosystems and programming languages. The small package size in the npm ecosystem could lead to different dynamics compared to other ecosystems, which might significantly affect packages' characteristics such as the maintenance lifetime, release span, and barriers to migrate to other packages. While centrality is a commonly employed metric to evaluate the importance in highly-connected systems, such as software ecosystems, the performance of our approach might be linked to the highly dynamic characteristics of npm. Future work needs to investigate if a similar approach can also help identify packages in decline in other ecosystems such as PyPi and Maven.

## **6.8 Chapter Summary**

This chapter presents a novel and scalable approach for using the centrality of packages to identify packages in decline. Our evaluation showed that the centrality trends were effective at identifying packages in decline (RQ1). When classifying packages as in decline

and not in decline, our approach can distinguish between the two classes with an AUC of 0.9. Our approach correctly classified 87% of the packages in decline, on average 18 months before the *npm*s aggregated score (RQ2). By evaluating the correlation between centrality and current popularity metrics (e.g., number of downloads), we have shown that centrality trends can provide new information, not currently captured by *npm*s (RQ3). We implemented our approach in a tool that can be used by developers to complement current *npm*s popularity metrics with our centrality trends. Our approach can provide a more accurate depiction of the shifts the community interest makes and help inform developers when selecting packages for their software projects.

Our work outlines some directions for future work. First, in this chapter, we use centrality as an indicator of packages in decline. We believe investigating and understanding why packages' centrality is rising or declining is critical because it helps developers make more informed decisions. Another interesting followup work is to propose an automated approach to finding future central packages so they can receive the attention needed to boost their evolution as early as possible. Finally, after identifying packages in decline, the next step should be assisting developers in replacing them. Thus, in the next chapter, we propose an approach that finds alternative packages for those in decline.

# Chapter 7

## An Approach to Find Alternative Packages

### 7.1 Introduction

Since software, like people, get old (Parnas, 1994), developers need to keep up with the changes in the ecosystem to avoid depending on packages that became obsolete, dormant, or even deprecated (Valiev et al., 2018). Community interest uphold packages to improve, i.e., include better features driven by community needs, keep up the package maintenance by reporting bugs to maintainers, motivate maintainers to continue supporting the package, and some times even financially support the maintainers on platforms such GitHub Sponsors,<sup>1</sup> Open Collective,<sup>2</sup> and Tidelift.<sup>3</sup> Packages that show a decline in community interest are usually used less over time, become less frequently maintained, and eventually, could become abandoned (Khondhu et al., 2013; Valiev et al., 2018). Moreover, a package's decline in community interest may indicate that a better solution is drawing attention in the ecosystem, and developers are migrating to a package that better suits their needs.

---

<sup>1</sup><https://github.com/sponsors>

<sup>2</sup><https://opencollective.com>

<sup>3</sup><https://tidelift.com>

Prior work examined projects that are unmaintained (Coelho et al., 2020, 2018). In Chapter 6, we proposed an approach identified packages that lose popularity over time (i.e., are in decline). Other studies proposed approaches for mining dependency migrations from software repositories to suggest alternatives (Alrubaye et al., 2019a; He et al., 2021; Teyton et al., 2012, 2014). However, to the best of our knowledge, little attention has focused on suggesting alternatives to packages that are in decline, especially in the context of dynamic programming languages such as JavaScript.

Therefore, in this chapter, we leverage the crowd wisdom in the software ecosystem to suggest alternatives to packages that are in decline. Our approach uses dependency migrations from real-world projects to identify alternative packages. Moreover, our approach suggests dependency migrations based on the community interest of the packages. Thus, our approach suggests replacing the packages that are in decline with alternative packages that still maintain the community interest.

We evaluate our approach on the npm ecosystem, the largest growing ecosystem to date (Decan et al., 2019), and the host of JavaScript reusable packages, currently the most popular programming language (*Stack Overflow Developer Survey 2018*, 2018). The popularity and scale of the npm ecosystem make it an ideal candidate for our study. We evaluate the accuracy and the usefulness of our approach in generating alternatives for packages in decline, through the following three research questions:

**RQ1: How accurate is our approach to suggest alternative npm packages?** (Section 7.3) We manually evaluate the accuracy of our approach in suggesting alternative packages that perform comparable functionality. We found that our approach provided valid alternative packages in 96% of the generated suggestions.

**RQ2: How useful is our approach to JavaScript project maintainers?** (Section 7.4) We survey JavaScript developers to assess the usefulness of our approach. We found that our approach provided new information about alternative packages for 54% of

the developers. On a 5-points Likert scale, developers recommend having a tool that utilizes our approach to suggest alternative packages with median = 4. More importantly, 67% of the developers confirmed that they would use our suggested alternative packages in their projects.

**RQ3: When and why maintainers migrate to depend on the alternative npm packages?** (Section 7.5) We manually examine pull requests that perform dependency migrations that match the generated suggestions by our approach. We found that the majority of migrations (69%) occurred during dependency management tasks, however, 31% of migrations occur during other development tasks, i.e., fixing bugs (16%), adding new feature (8%) and code refactoring (7%). Also, we found that the primary motivation (74% of the cases) of dependency migrations was to replace unmaintained dependencies.

Our findings show that our approach is accurate and helpful to JavaScript developers. The following are the key contributions of our work:

- Propose an approach to suggest alternatives for packages in decline.
- Empirically evaluate our approach accuracy on the npm ecosystem.
- Surveyed expert JavaScript practitioners to assess the usefulness of our approach.
- Illustrate the characteristics of the dependency migrations suggested by our approach.
- Support the replication and future research by making all of our datasets (i.e., collected data, analysis results, scripts) publicly available ([Mujahid, Costa, Abdalkareem, & Shihab, 2021b](#)).

The remainder of this chapter is organized as follows: Section 7.2 details our approach in suggesting alternatives for packages in decline. In Section 7.3, we evaluate the accuracy of our approach. Then, in Section 7.4, we survey JavaScript developers to assess the



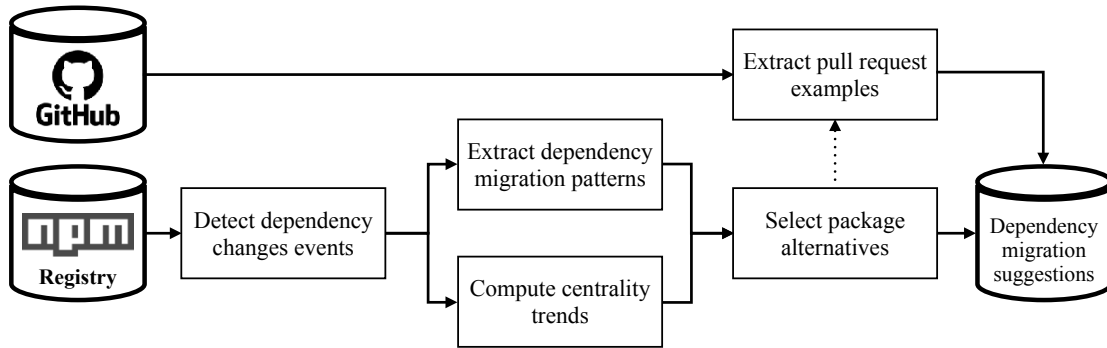


Figure 7.1: Our approach to suggest package alternatives.

usefulness of our results. Section 7.5 presents the characteristics of the suggested package alternatives. In Section 7.6, we describes the threats to validity. Finally, we present the conclusion of our work in Section 7.7.

## 7.2 Approach

In this section, we explain our approach that uses the dependency migration patterns in the npm ecosystem and the packages’ centrality trends to suggest alternative packages for the ones that are in decline. Figure 7.1 shows our overall approach, which starts by detecting dependency change events from the npm ecosystem. Then, we analyze these events to extract dependency migration patterns and calculate the centrality trends for all packages in the npm ecosystem. Next, we filter the dependency migration patterns to select migration suggestions that are more likely to recommend better alternative packages. Finally, we extract pull request examples that match the dependency migration suggestions generated by our approach.

## 7.2.1 Detect Dependency Change Events

The core of our approach lies in identifying frequent dependency migration patterns occurring in the ecosystem to better inform or recommend practitioners. To extract dependency migration patterns or compute the centrality trends, we need to detect *dependency change events* in the npm ecosystem. In our study, we consider two events as dependency change events: 1) the addition of a new package dependency and 2) the removal of a package dependency. Hence, dependency change events do not include updating the version of a dependency, since our approach aims to suggest dependency migrations regardless of their versions. Later in our process (in Section 7.2.2), we use dependency change events to detect the dependency replacements, i.e., in which one dependency is removed and another one is added. Also, we use the dependency change events to update the npm ecosystem's dependency graph and calculate the centrality trends (in Section 7.2.3).

To extract dependency change events for all packages in the npm ecosystem, we analyze the entire npm registry database. The npm registry maintains a record of the packages' dependencies for each version of every package. For each package in the registry, we start by sorting the package versions in ascending order by their release time. Then, on each version ( $v_n$ ), we compare the list of dependencies with the previous version ( $v_{n-1}$ ). If the dependency is absent in the version  $v_{n-1}$ , we consider it to be a dependency addition event; conversely, if the dependency is absent in the version  $v_n$ , we consider it a dependency removal event.

It is crucial to notice that package releases can be nonlinear. Package maintainers commonly employ backports to fix older release versions (Decan, Mens, Zerouali, & De Roover, 2021). In such cases, the chronological order of the versions will be polluted by backports, as these versions could include old dependencies no longer used in the main release branch. Hence, in our process, we filter out any release with a lower semantic versioning than its predecessor in relation to their respective release date. For example, the developers of

the package `react` have released the version `16.13.1` in March 2020, then in October 2020 released the backport version `15.7.0` to fix an older major version, where the latest version was `15.6.2` (Inc., 2121). Since the version `15.7.0` is smaller than the version `16.13.1`, we exclude the version `15.7.0` from our analysis.

## 7.2.2 Extract Dependency Migration Patterns

We extract dependency migration patterns by identifying recurring *dependency replacements* in the npm ecosystems. First, we use the dependency change events to retrieve changes in the packages' dependencies across all their versions. Then, we consider each of the added dependencies as a potential replacement for each of the removed dependencies. Listing 7.1 show an example of dependency changes in a `package.json` file between two package versions. The dependency changes in the example are represented as four dependency change events. Three dependencies are removed (i.e., removal change events), and one dependency is added (i.e., an addition change event). By applying this on the example in Listing 7.1, we extract three dependency replacements: `less`  $\rightarrow$  `lodash`, `underscore`  $\rightarrow$  `lodash` and `utf-8-validate`  $\rightarrow$  `lodash`. In our process, we do not mix runtime dependencies with development dependencies. Thus, we consider the added runtime dependencies as potential replacements for only the removed runtime dependencies, not the development dependencies, and vice versa. We do this since the development and runtime dependencies are typically used in different contexts and should not be recommended as alternatives to each other.

When a package releases a new version and replaces more than one dependency, our approach has no way to identify which dependency has been replaced nor its replacement. Thus, as explained earlier, we consider each added dependency a potential replacement for each deleted dependency (combination). Since we want to reduce the odds of combinatorial explosion of dependency replacements, we filter out releases with massive or imbalanced

Listing 7.1: Dependency changes taken from the version 0.2.16 of the package @jpmorganchase/perspective.

```
"dependencies": {
  "@babel/polyfill": "^7.0.0",
  "@babel/runtime": "^7.1.5",
  "bufferutil": "~3.0.0",
  "d3-array": "^1.2.1",
  "detectie": "1.0.0",
  "flatbuffers": "^1.10.2",
-  "less": "^2.7.2",
+  "lodash": "^4.17.4",
  "moment": "^2.19.1",
  "papaparse": "^4.3.6",
  "text-encoding-utf-8": "^1.0.2",
  "tslib": "^1.9.3",
-  "underscore": "^1.8.3",
-  "utf-8-validate": "~4.0.0",
  "websocket-heartbeat-js": "^1.0.7",
  "ws": "^6.1.2"
},
```

number dependency changes. We only consider dependency change events from releases where the difference between the number of added dependencies ( $D_a$ ) and removed dependencies ( $D_r$ ) are close, i.e.,  $|D_a - D_r| \leq 1$ . Also, we avoid considering change events where there is a large number of added and removed dependencies, i.e.,  $D_a + D_r \leq \tilde{x}$ , where  $\tilde{x}$  is the median value for  $D_a + D_r$  across all the releases in the npm registry. We do this filtering since a large number of dependency changes indicates a significant code refactoring more than a simple dependency replacement.

We consider a *dependency migration pattern*, a dependency replacement that frequently occurs in the ecosystem, which is more likely to indicate a trend. Hence, after identifying the dependency replacements, we consider replacements that reoccur at least 10 times in our dataset as dependency migration patterns. That is, at least the developers of 10 distinct projects must have performed the same dependency replacement.

### 7.2.3 Calculate Centrality Trends

Centrality has been used as a proxy of community interest, where packages that show a decline in centrality are usually used less over time, become less frequently maintained, and eventually could become abandoned. Thus, our approach uses the centrality to target suggesting alternatives for packages that are in decline to replace them with packages deemed not in decline. To determine if a package is in decline or not, we use the approach that we proposed Chapter 6 which requires dependency change events to calculate the centrality and detect packages in decline. A package considered in decline if it shows statistically significant declines in the centrality over a specific period of time.

Our approach requires the centrality trends for packages engaged in the extracted dependency migration patterns. However, calculating the centrality rankings requires computing the centrality for every package in the ecosystem. Thus, we use the dependency change events (extracted in Section 7.2.1) to calculate the monthly centrality rankings for each package in the npm registry.

### 7.2.4 Select Package Alternatives

Once we have both the dependency migration patterns (Section 7.2.2) and centrality trends for all packages (Section 7.2.3), we select the most promising dependency migration patterns to recommend to practitioners. To do so, we use the following criteria:

- **The replaced package is in decline:** we select only patterns where the removed package is in decline. Since the decline can vary based on the examined period, we measure the decline over three different periods: the last six months, the last year, and the package's overall lifetime. If the package shows a decline based on one of the measured periods, we consider it an in decline package.

- **The alternative package is not in decline:** to ensure that we suggest better alternatives, we select only patterns where the added package is not in decline.
- **The migrations pattern is performed recently:** to avoid recommending outdated migration patterns, we considering only the patterns performed at least once in the last 90 days.
- **Performed by a popular project:** to avoid considering patterns performed only by immature projects, a migration pattern should be performed by a popular project in order to be considered. In this context, we consider a project as popular if the project is in the top 10% of most central packages in the npm ecosystem.

When our approach finds more than one package alternative, we select the dependency migration pattern with the highest support, i.e., performed more frequently.

## 7.2.5 Extract Pull Request Examples

We aim to provide developers with examples of pull requests that performed the suggested dependency migration. Exemplary pull requests may provide insights on the migration's efforts, reasons for the dependency migration, and help practitioners understand the differences between the alternatives. To extract pull requests examples, we check the npm registry to find packages that performed any of the candidate dependency migration patterns. For these packages, we collect their repository addresses on GitHub. Then, we use the GitHub API<sup>4</sup> to extract all the merged pull requests from the selected repositories.

Once we retrieve the pull requests from a repository using the GitHub API, we select only the pull requests that perform the suggested dependency migrations. To do so, we consider only the pull requests that modify a `package.json` file, which is the file where projects declare their dependencies. Then, we compare the content of the `package.json`

---

<sup>4</sup><https://docs.github.com/graphql>

file as it is on the merge commit with the content of the files as it was on the parent commit. Next, we exclude pull requests that are extremely big, i.e., change more than 100 files, which is 7 times more than average number (mean = 13.62) of changed files in pull requests (Gousios & Zaidman, 2014).

## 7.3 Accuracy of the Approach

In this section, we investigate the performance of our approach in suggesting alternative packages for those in decline. The decline of package centrality could be a symptom that better alternatives have emerged or a shift in the community interest. However, developers have little information to grasp where the community has shifted its interest. Suppose our approach can aptly capture valid package alternatives. In that case, it can be embedded in dependency management tools, such as the npm CLI, to increase developers' awareness of alternative packages and help them reevaluate their package dependencies.

To measure the accuracy of our approach, we first use our approach to generate dependency migration suggestions to alternative packages. Then, manually evaluate whether the suggested alternative packages perform comparable functionalities to the original ones.

### 7.3.1 Generate Dependency Migration Suggestions

We generate the suggestions to alternative packages using the approach described in Section 7.2. We start by detecting the dependency change events as of December 22, 2020. We collected in total 18,459,923 dependency change events from 1,148,720 packages from the npm ecosystem. From these change events, we extracted 2,434 dependency migration patterns. After filtering the migration patterns based on the centrality trend of the packages and the criteria described in Section 7.2.2, we end up with 152 package alternative suggestions.

### 7.3.2 Manual Evaluation Process

Once we generate the dependency migration suggestions, we manually evaluate the correctness of the suggested alternative packages by assessing whether both packages provide similar functionalities or not. We manually examine the documentation (e.g., readme file, homepage, and website) of the package to be replaced and the suggested alternative package to understand their functionalities. We start by inspecting the documentation of the package homepage on the official npm website<sup>5</sup>. If the description is not descriptive enough for our classification, we examine other available sources such as the readme file, the package website, and the package repository. Once we have a comprehensive understanding of both packages, if the suggested alternative package performs comparable functionalities to the original package, we consider the suggested alternative package as a valid alternative package. For example, the packages `commander` and `yargs` share similar functionalities, helping in building command-line interfaces for Node.js. Hence, we consider the package `commander` as an alternative for `yargs`.

In total, two of the authors independently examined the documentations of 256 packages for 152 dependency migration suggestions. Since this process involves human judgment, it is prone to human bias. We assess the agreement of both examiners using the Cohen-Kappa inter-rater reliability. Cohen-kappa inter-rater reliability is a well-known statistical method that evaluates the inter-rater reliability agreement level. The result is a scale that ranges between -1.0 and 1.0, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement (McHugh, 2012). In our analysis, we found that both authors have an excellent agreement (kappa=0.79). After the initial classification, any disagreement was examined and discussed by both examiners to reach consensus (Fleiss & Cohen, 1973).

---

<sup>5</sup><https://www.npmjs.com>



Table 7.1: Summary of the suggested alternatives categorize by their functionalities.

Category	Suggestion Example	TP	FP
Building	<code>browserify</code> → <code>webpack</code>	39	3
Utilities	<code>moment</code> → <code>dayjs</code>	26	-
Testing	<code>vows</code> → <code>mocha</code>	16	-
User Interface	<code>jade</code> → <code>pug</code>	12	-
Linters	<code>jshint</code> → <code>eslint</code>	10	2
Client Library	<code>redis</code> → <code>ioredis</code>	10	1
Networking	<code>request</code> → <code>axios</code>	10	-
Parser	<code>esprima</code> → <code>acorn</code>	9	-
CLI	<code>optimist</code> → <code>yargs</code>	4	-
Other	<code>memory-fs</code> → <code>memfs</code>	10	-
<b>Total</b>		<b>146</b>	<b>6</b>

### 7.3.3 Results

Based on the manual evaluation of our approach, Table 7.1 shows a summary of the migration suggestions generated by our approach. We categorize and group the suggestions by their abstracted functionalities. Each category of suggestions in the table has an example of a package migration suggestion generated by our approach. The third column presents the number of True Positive cases (TP), where our approach accurately finds the alternative packages. The last column shows the number of False Positive cases (FP), where our approach suggested invalid alternatives.

Overall, we examine 152 dependency migration suggestions generated by our approach. We found that 146 (96%) of the generated dependency migration suggestions include valid alternative packages and 6 (4%) of them do not. Furthermore, we found that the packages performing functionality related to building the JavaScript projects have the highest share (28%) of the generated dependency migration suggestions. For example, our approach suggests replacing the JavaScript bundler package `browserify` with a more popular, scalable, and feature rich one, the `webpack` package. The next category in our results is the utility tools (17%), where our approach suggests replacing obsolete utility packages

such as `moment` with a more modern solution like `dayjs`. Next categories include testing tools, and user interface components and helpers with share of 11% and 8% respectively. Also, among others, the categories include linters to enforce rules on the JavaScript code, clients drivers to interact with other services, and networking utilities.

Out of the 6 invalid dependency migration suggestions that we found, only one invalid suggestion belongs to replacing runtime dependency, where the remaining 5 cases suggest replacing development dependencies. An example of invalid dependency migration suggestion, is the suggestion of replacing the development dependency `ember-cli-eslint` with `eslint`. However, `ember-cli-eslint` is a plugin to identify and report patterns found in Ember<sup>6</sup> projects using the package `eslint`, which we consider an invalid alternative package of `eslint`.

Since runtime dependencies and development dependencies are treated differently by developers, we separate the results of our tool performance based on the dependency type. Interestingly, we found that the accuracy of our tool in suggesting package alternatives for development dependencies is 93%, where it is 99% for development dependencies.

**Summary of RQ1:** Out of the 152 dependency migration suggestions generated by our approach, we found that 96% of them are valid alternative package suggestions. Most frequent suggestions recommend alternatives for building, utilities and testing packages.

## 7.4 Usefulness of the Approach

After finding our approach provides accurate recommendations, we evaluate if practitioners find our suggestions practical. More specifically, we want to know if our approach

---

<sup>6</sup><https://emberjs.com>

presents new information to practitioners: Are practitioners aware of the suggested migrations? Do practitioners believe our approach is valuable? Would practitioners act upon the suggested changes and migrate packages in their own projects?

To evaluate the usefulness of our suggested package alternatives, we conducted a survey to collect feedback from JavaScript project maintainers. In the following, we will present our survey design, participant recruitment process, the background of the participants, and the survey results.

### **7.4.1 Survey Design**

We design a survey to evaluate the usefulness of the suggested dependency migration to JavaScript developers that use the packages in decline in their projects. That is, we target our survey to developers that have used the packages our approach recommends replacing. Our survey contains three main parts. We ask JavaScript practitioners about: 1) their software development background, 2) if they are aware of the suggested alternatives, and 3) their perceptions about our results. Table 7.2 shows the questions we ask along with the type of accepted answers for each question.

In the first part, the survey contains questions related to the surveyed participants' experience and background. We ask these questions to ensure that our survey participants have sufficient experience using npm packages in software development and maintenance. In the second part of the survey, we ask the participants whether they know the suggested package alternative and are aware of the JavaScript community's migration trend toward the suggested alternative package. We inform the participants about the dependency migration suggestion in the survey invitation email.

In the last part of our survey, we want to understand the participants' perceptions about

Table 7.2: Questions in our survey about the alternative package suggestions.

Category	Question	Accepted Answers
Background	How would you best describe yourself?	<i>Single selection options:</i> Full-time, Part-time, Free-lancer, or Other.
	For how long you have been developing software?	<i>Single selection options:</i> Less than 1 year, 1 to 3 years, 4 to 5 years, or More than 5 years.
	How many years of JavaScript development experience do you have?	
	How many years of experience do you have using the Node Package Manager (npm)?	
	How often do you search for npm package alternatives?	<i>Single selection options:</i> Never, Rarely (e.g., once a year), Sometimes (e.g., once a month), Often (e.g., once a week), or Very often (e.g., everyday).
Awareness	Are you aware of the alternative package mentioned in the email?	<i>Single selection options:</i> Yes or No.
	Are you aware of the JavaScript community's migration trend mentioned in the email?	
Usefulness	Do you think that a tool that helps generate potential alternative packages would be useful?	<i>Likert-scale:</i> ranges from 1 = Not useful, to 5 = Extremely useful.
	Do you believe that providing an example of Pull Requests of migrations from other projects would be helpful?	<i>Multiple selection options:</i> Help in estimating the dependency migration efforts, Help in justifying the dependency migration, Help in understanding the required API changes, Not helpful, and Other.
Future Actions	In your future new projects, will you use the alternative package?	<i>Single selection options:</i> Yes or No.
	If the previous answer was "No", why not?	<i>Free text</i>
	In your current projects, will you advise to migrate to the alternative package?	<i>Likert-scale:</i> ranges from 1 = Keep the current package, to 5 = Strongly advise migrating.

our suggested dependency migrations. Thus, we ask the participants two groups of questions: 1) how useful they found our suggested dependency migrations and 2) their willingness to take actions related to our suggestions. In the first group (two questions), we ask if the participants think there is a need for a tool to generate suggestions for alternative packages. Also, since our approach provides real-world dependency migration examples in the form of pull requests, we asked how helpful was the provided pull request examples

in evaluating the dependency migration. In the second group (two questions), we ask if the participants will use the suggested alternative package in their future projects. If the answer is No, we request the participants to explain why. We ask this open-ended question to give our survey participants maximum flexibility to express their opinion and experience as recommended in survey design guidelines (Dillman, 2011). In addition, we asked the participants if they think their current projects should migrate to use the suggested alternative packages.

## 7.4.2 Participant Recruitment

To select our survey participants, we reach out to experienced developers who have adopted the packages that we suggest to be replaced with alternatives. We retrieve a list of JavaScript projects hosted on GitHub that have at least 100 stars. We use the project stars as a filtering criteria, as commonly done in the related literature (Abdalkareem, Oda, et al., 2020; Borges et al., 2016; Golubev, Eliseeva, Povarov, & Bryksin, 2020), and by selecting highly-starred projects we mitigate the chances of including immature and personal projects in our survey. We were able to retrieve a list of 35,719 projects using the GitHub search API.<sup>7</sup>

Next, we use the GitHub raw content API<sup>8</sup> to retrieve the list of dependencies from the `package.json` file of each project. If a project has any dependencies that our approach suggests to be replaced, we clone the project’s repository to be analyzed. To select participants with experience in using the dependencies that our approach suggests replacing, we target the developers who introduced these dependencies in their projects. Thus, we use `git`<sup>9</sup> to retrieve the change history of the `package.json` file. On each commit that modifies the `package.json` file, we detect the dependencies that were added. We do so

---

<sup>7</sup><https://docs.github.com/rest/reference/search>

<sup>8</sup><https://raw.githubusercontent.com>

<sup>9</sup><https://git-scm.com>

by comparing the dependencies as it appears in the examine commit and dependencies appears in the previous commit, i.e., its parent commit. If the commit is adding a dependency that our approach suggest to be replaced, we retrieve the author contacts from the commit metadata.

Since it is common for developers to use multiple email addresses (Zhu & Wei, 2019), we avoid selecting multiple contacts with the same name even if they have different email addresses. Also, to prevent distrusting teams with multiple survey invitations and to diversify our participants, we avoid selecting more than one developer per GitHub organization. Based on these steps, we were able to identify 4,696 unique JavaScript developers. Then, we randomly selected 1,000 developers to participate in our survey.

Finally, we send email invitations of our survey to 1,000 JavaScript developers. However, some email invitations were not able to be delivered, e.g., email address not found. Thus, we successfully delivered email invitations to 886 unique developers. As a result, we received 52 responses for our survey after having the survey open for a month, where we received all responses within the first two weeks. This number of responses translates to a 6% response rate, which is comparable to the response rate reported in other software engineering surveys (Buse & Zimmermann, 2012; Smith et al., 2013).

### **7.4.3 Survey Participants**

Table 7.3 shows the background of our survey participants, including the positions and experience of the participants, specifically, their experience in software development, JavaScript development, and the use of the npm packages.

Of the 52 participants in our survey, 41 participants identified themselves as full-time developers and only one as part-time developer. Also, 9 participants consider themselves as freelancers. As for the participants' experience, 47 participants have more than 5 years of development experience, where the remaining 5 participants have between 4 to 5 years. In

Table 7.3: Participants' position and experience in software development, JavaScript development, and using npm.

Developers' Position	Occurrences	Development Experience	Occurrences	Experience in JavaScript	Occurrences	Experience in Using npm	Occurrences
Full-time	42	1 - 3	0	1 - 3	3	1 - 3	3
Part-time	1	4 - 5	5	4 - 5	8	4 - 5	9
Freelancer	9	> 5	47	> 5	41	> 5	40

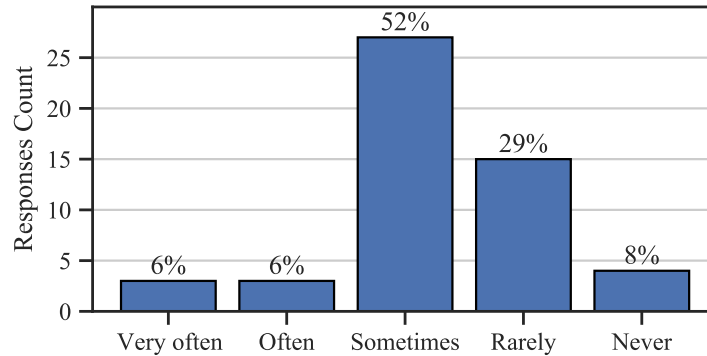


Figure 7.2: Survey responses regarding how often our survey participants search for package alternatives. In our survey, the question has the following answers: never, rarely (e.g., once a year), sometimes (e.g., once a month), often (e.g., once a week), very often (e.g., everyday).

addition, 41 participants have more than 5 years of experience developing using JavaScript, 8 participants claim to have between 4 to 5 years of JavaScript experience, and 3 participants claim to have between 1 to 3 years of JavaScript experience. We also asked our survey participants about their experience in using npm packages. Most of our survey participants (77%) have more than 5 years of experience using npm packages. Only 9 participants have between 4 to 5 years, and 3 have between 1 to 3 years of experience using npm.

Finally, we ask the participants how often they search for alternatives npm packages, to evaluate their interest and experience in searching for alternative packages Figure 7.2 show that out of the 52 participants, 92% of them do search for alternatives npm packages. Specifically, 6% of the participants search for alternatives npm packages very often, and another 6% claimed that they often search for alternatives npm packages. The majority of other participants (52%) state they sometimes look for alternative npm packages, and

29% mentioned that they rarely search for alternative npm packages, e.g., once a year. Interestingly, only 8% of our survey participants report that they never search for alternative packages.

Overall, the information from the background sections shows that all of our survey participants are experienced developers, and the vast majority of them have long experience in JavaScript and npm packages. Also, from the background sections, we see that most of our survey participants have experience searching for alternative npm packages. This gives confidence in the quality of our survey responses.

#### 7.4.4 Results

We measure our approach usefulness along three complementary dimensions: 1) developers awareness of the generated suggestions, 2) developers perceptions of our suggestions, and 3) developers willingness to take future actions based on our suggestions.

**Developer Awareness.** We assess the awareness about the generated suggestions by asking the participants 1) if they know the alternative package and 2) whether they are aware of the migration trend in the JavaScript community toward the alternative packages. Based on our survey responses, our approach were able to inform participants about the alternative packages and the JavaScript community's migration trend. Specifically, Figure 7.3 shows that 37% of the participants are not aware of suggested alternative packages, and 48% of the participants are not aware of the dependency migration trend in the JavaScript community. Given our suggestions are based on migrations that have been performed many times in the npm ecosystem, we found it surprising that 37% of participants have not heard of the alternative package before. To put things into perspective, all of our survey participants are familiar with the original package and 92% have reported to regularly look in the ecosystem for alternative packages. This indicates that even experienced developers are frequently unaware of the ecosystem's trends and need tools to be better informed about



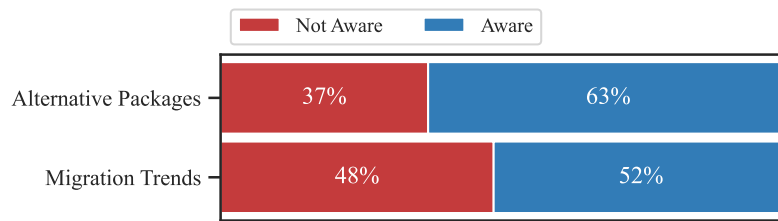


Figure 7.3: The awareness of participants about the suggested alternative packages and the community’s migration trends.

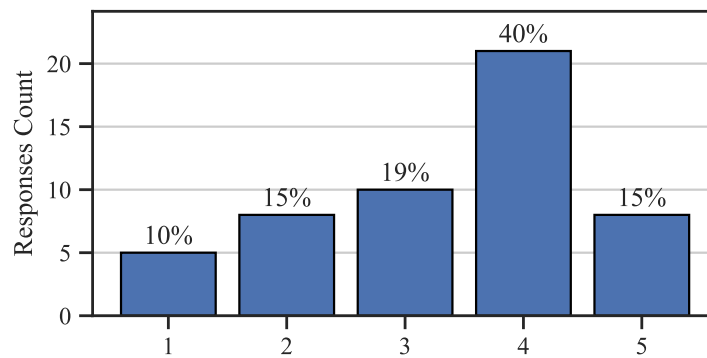







Figure 7.4: Survey responses regarding the usefulness of having a tool that generate alternative suggestions using our approach. The usefulness rated on a 5-points Likert-scale ranges from 1 = Not useful, to 5 = Extremely useful.

their development community.

**Developer Perceptions.** To understand the perception of the project maintainers of our approach’s results, we ask the participants if they think that a tool that helps generate potential alternative packages would be useful. As Figure 7.4 shows, most participants believe that the suggestions generated by our approach are useful and support the idea of having a tool to generate such suggestions. On a 5-points Likert scale, the support of such a tool has a median = 4 and mean = 3.37, where only 10% of the participants claimed it is not useful for them, and 15% indicate that it is extremely useful.

In the survey invitation, we provide the participants with pull request examples of dependency migrations from other projects. We ask the participants if they found the pull request examples helpful. In this question, participants can select more than one option or

Table 7.4: Participants’ responses on how helpful are the examples of dependency migrations from other projects?

Helpfulness	Frequency
Understanding the required API changes	79% 
Estimating the dependency migration efforts	75% 
Justifying the dependency migration	52% 
Other	6% 
Not helpful	11% 

even enter their own answer. Table 7.4 shows the participants’ responses. We can observe that the majority of the participants (88%) believe that the provided examples of dependency migrations from other projects are helpful. Specifically, 79% of the participants find that the provided examples are helpful in understanding the differences and the required changes in the API usage between the alternative package and the current package. Also, 75% of the participants indicate that the migration examples from other projects can help in estimating the efforts needed to migrate to the alternative package in their projects. Interestingly, 52% of the participants mentioned that the explanation in the provided examples helps justify the dependency migration within their teams. Finally, only 12% of the participants claim that the provided dependency migration examples are not helpful. For example, participant P3 expressed a disagreement with the justification given in the pull request example. The participants believe that the alternative package has more dependencies, which is the opposite to what was explained in the pull request example, “Just reading briefly, but yargs has way more dependencies than commander, contrary on what is reported in the PR”. Another participant mentioned that a pull request example would only be useful to them if it illustrates solving a security issue (P50: “I would never bother migrating unless there were a severe and applicable security concern ...”). Also, participant P42 states “Helpful in giving an idea of the change, but not much beyond that (every project likely requires custom things).”

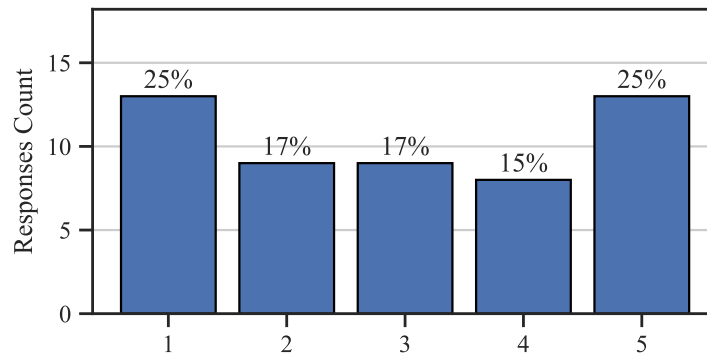


Figure 7.5: Survey responses regarding the support of migration their current projects to use the alternative packages. The support rated on a 5-points Likert-scale ranges from 1 = Keep the current package, to 5 = Strongly advise migrating.

**Future Actions.** Whether developers would act upon our suggestions is a strong indication of our approach’s usefulness. Thus we ask the participants 1) if they plan to use the alternative packages in their future projects and 2) how likely are practitioners to migrate to alternative packages in their current projects. The responses from our survey shows that 67% of the participants will use the alternative packages in their future projects. However, participants are split between supporting the dependency migrations and keeping current packages in their current project. Figure 7.5 shows the distribution of the participants’ responses. On a 5-point Likert-scale, participants rate their support to migrate to the alternative package in their current projects in median = 2 and mean = 2.98.

To understand why participants opt for retaining current packages, even if the package has been declining in the community, we analyze the free-text justifications from our survey participants on why they will not use the alternative package. In total, we manually examined 17 responses (out of the 52 responses) from the participants who indicated that they would not use the alternative package. Based on this analysis, we observe different reasons why they will not use the alternative package. Six participants mentioned that they would not use an alternative package as long as the current one is working. As one respondent P2 summarizes “If it ain’t broke, don’t fix.” While migrating to a dependency

with more functionalities can be helpful, the benefits of migrating will only be achieved if the project needs the extra functionality. One of our participants (P47) expressed this concern and said “The alternative packages bundles many functions, I need just the one that the old package uses.” In contrast, P3 prefers not to use a package that has more dependencies, and state “Because I value packages with little to none dependencies, one reason (of many) is the Dependabot’s alerts hell, which is a huge waste of time, more often than not.” Other participants indicated having less priority to migrate packages that perform a minor functionality or packages that they do not use in the production. For example, P9 said “It’s just a small development time dependency used during build...as long it works - it works.” Interestingly, some participants will not use the alternative packages because their current projects have low priority. For example, P13 said “The project is mostly deprecated and will be moving to a new go-lang based system ...” Also, P28 said “... that was for an open source project, I already need to spend time with user issues and such, so using the shiny latest version of a package is not really what I want to spend my free time on. For a paid/work project, that would be different.”

**Summary of RQ2:** On a 5-points Likert scale, the developers support having a tool that utilizes our approach to suggest alternative packages with median = 4 and mean = 3.37. Out of our 52 participants, 67% will use our suggested alternative packages in their future projects. Moreover, our approach helped 48% of the participants learn about the community trend toward migrating to the alternative packages.

## 7.5 Characteristics of the Suggestions

We are also interested in understanding *when* and *why* developers do the kind of package dependency migrations that our approach suggests. Answering the question *when* aims to discover the type of maintenance activities that developers are performing when they

migrate to the alternative packages. While answering the question *why* targets finding out the reasons that motivate the developers to migrate to the alternative packages. Answering both questions will provide insights on how to employ our approach in the development process e.g., development tools and CI pipelines.

To understand the developers' behavior about performing dependency migrations to replace packages, we manually classified pull requests that perform dependency migrations that match the migration suggestions by our approach.






### **7.5.1 Manual Classification**

We start by extracting pull requests that perform dependency migrations that match the dependency migration suggestions generated by our approach. To this aim, we utilize the approach described in Section 7.2.5. As a result, we obtain a list of 225 pull requests that perform dependency migrations from 155 different GitHub repositories.

Once we have the dependency migration pull requests, we perform an iterative coding process to classify and group pull requests. We gradually develop two sets of codes based on an inductive analysis approach (Seaman, 1999). The first set of codes concerned the purpose of the pull request (activity) and the second set of codes concerned the motivation of the dependency migration performed in the pull request.

In the process of classifying the activity of a pull request, we focus on the primary goal of the pull requests, as described in its title and description. Thus, we tag each pull request with only one activity type. Also, we examine the pull request description and discussion to find the motivation of the dependency migration. However, not all pull requests include an explanation to justify the dependency migration. For the ones that have a justification, we tag them with the migration motivation. Two authors independently tag each of the 225 pull requests with a single activity type, and applicable pull requests are also tagged with the migration motivation.

Table 7.5: The activities of the pull requests that performed the dependency migrations.

Activity	Description	Frequency
Dependency update	Intended mainly to update dependency versions.	42% 
Dedicated migration	The main goal is to migrate to a different package.	27% 
Bug fixing	Aims to resolve issues in the project.	16% 
New feature	Adds a new functionality or feature to the project.	8% 
Refactoring	The objective is refactor existing code.	7% 





As with any other manual classification activity, there is some level of subjectivity that may generate disagreement between the annotators. To account for this, we applied a Cohen’s Kappa to measure the level of agreement between the two individual classifications (Cohen, 1960). In our analysis, we found that both authors have an excellent agreement (kappa=0.93) on classifying the pull request activates. Also, the authors have an excellent agreement (kappa=0.90) on classifying the motivation of the dependency migrations.

## 7.5.2 Results

Based on the manual classification of the pull requests, we organize the results into two parts. The first part discusses the type of maintenance activities that were performed when developers migrate their dependencies. The second one discusses the motivation of the dependency migrations.

**When developers perform dependency migrations?** Table 7.5 shows our classification results of the pull request activities. We found that 42% of the pull requests perform dependency migrations as part of a dependency update. For example, in one of the projects, the developer created a pull request to update multiple dependencies, and in the same time migrated from using the package `node-uuid` to `uuid` (Bernhardt, 2019). Interestingly, we found that only 27% of the pull requests are dedicated to performing a dependency migration activity. In a dedicated migration, the main goal of the pull request is to just

Table 7.6: The motivations of 62 pull requests that performed the dependency migrations.

Motivation	Description	Frequency
Maintenance	Better quality and better maintained alternative.	74% 
Compatibility	Increase compatibility with other packages or systems.	15% 
Performance	Faster execution, less dependencies, or smaller bundle size.	8% 
Features	Providing missed features or flexible API.	3% 

replace a dependency with another. In contrast, a pull request tagged with dependency update activity aims to update the version of one or more dependencies, but it replaces other dependencies in the same pull request.

From our analysis, we identified that 16% of the dependency migrations were a part of a bug fixing activity. For example, in one of the projects, a developer created a pull request to fix a bug by migrating to an alternative package (Quixada, 2017). He describes the issue as the following: “isomorphic-fetch has a bug that prevents it from running in a react native environment. Since it is no longer maintained, it will never be fixed. That also means dependencies are outdated. cross-fetch is React Native compatible.” In 8% of the cases, we observe that the dependency migration occurs when developers add a new feature. An example of such a case, a developer migrated to an alternative package to support the out of the box installation on more platforms (Pakers, 2019): “And I am not arguing that it is not possible to install keystone on Windows, but this is far from a simple npm install. And as bcrypt is the problem, switching to bcryptjs would make it easier.” In the remaining cases (6%), we notice that the dependency migration was a part of refactoring activity. For example, in on of the projects, a developer performed a major refactoring to use plain TypeScript locally, in the same pull request, he migrated from depending on the package `rimraf` to the package `del-cli`.

**Why developers migrate the dependencies?** Our manual assessment found that 62 (24%) of the 225 pull requests provide an explicit justification of the dependency migration. In Table 7.6, we show the results of our manual classification.

In 74% of the cases, dependency migrations are motivated by the need for better maintained alternative packages. Even when the package is not officially deprecated, developers migrate from packages that are not well maintained, for example (Silbermann, 2020): “Removes isomorphic-fetch from the dependencies which doesn’t seem to be maintained anymore.” Even when a package has maintenance activities, developers were not satisfied with the quality level of the maintenance, one of the developers said (Hardcastle, 2019): “Sentry is moving away from the Raven library we are using and while it says it’s maintained, not every feature is working anymore.”

The second most frequent motivation for dependency migrations in our dataset is the compatibility with other packages or systems (15%). In these cases, developers migrate to use alternative packages that are more compatible with other project dependencies or to support more systems and platforms. For example, in one of the pull requests the developer migrated from the package `uglifyjs` to the package `terser` to be compatible with the new version of `webpack` package (Parsa, 2018): “Webpack requires `uglifyjs-webpack-plugin@1.x`. thus `uglifyjs-webpack-plugin@2.x` may not resolve correctly. Also, the webpack team decided to go with `terser-webpack-plugin`.” In another case, the migration performed to improve the support for the Windows operation system (Pakers, 2019): “It is a nightmare to run this project on Windows as it uses `bcrypt`.”

We observe that 8% of the dependency migrations are motivated by improving the performance of the project. Performance metrics mentioned include faster execution time, smaller bundle size shipped to production, and lower number of transitive dependencies. For example, one of the developers improved the performance by migrating from the package `moment` to the package `dayjs` (Waterloo, 2020): “`dayjs`, after webpacking, is about 7 KB compared to about 700 KB for `moment`. This will mean faster load times and smaller packed VSIXs.”

In the remaining of the pull requests (3%), we observe that the motivation is to use



features offered by the alternative packages. For example, a developer migrated from `sanitize-html` to `dompurify` in order of allowing for more HTML tags after sanitizing HTML content (Slagle, 2016): “`dompurify` prevents XSS but allows more tags and attributes than our previous sanitizer.”

**Summary of RQ3:** The majority of migrations (69%) are performed during dependency management activities, where 27% of the pull requests are dedicated to dependency migration. 31% of the pull requests perform dependency migrations alongside other activities such as bug fixing (16%), adding new features (8%), and refactoring (7%). Also, we found that 74% of the dependency migrations are performed to move to well-maintained alternative packages.

## 7.6 Threats to Validity

In this section, we discuss threats to the internal and external validity of our study.

### 7.6.1 Threats to Internal Validity

Threats to internal validity are related to experimenter bias and errors. A limitation of our approach is that it only considers dependencies between packages in the npm registry. This limitation will affect the centrality and migration patterns of packages that are not meant to be used by other packages, but other JavaScript applications, i.e., top-level packages. However, previous work has shown that using the npm registry as the sole source of changes in the dependency graph can serve as a proxy for the overall (Cogo et al., 2019; Cogo, Oliva, & Hassan, 2021). Future work should investigate how to incorporate JavaScript applications on the generated suggestions.

Another threat concerns the process we used to filter dependency change events to our approach. To reduce noise, we opted to remove dependency change events from massive

or imbalanced number dependency changes, as explained in Section 7.2.1. This may affect our recommended patterns, as they will more likely be based in projects that perform small dependency changes over time. We mitigate this effect by only considering migration patterns that reoccur across many projects.

Finally, our approach may contain bugs that may have affected our results. We made our scripts and dataset publicly available to be fully transparent and have the community help verify (and improve) our approach ([Mujahid, Costa, Abdalkareem, & Shihab, 2021b](#)).

## **7.6.2 Threats to External Validity**

Threats to external validity are related to the generalizability of our findings. Our evaluation focused entirely on the npm ecosystem, which has very particular characteristics: a centralized package registry, hundreds of thousands of software packages, and a very active and popular programming language. Also, packages in the npm ecosystem are relatively small compared to modules and software components in other ecosystems and programming languages. The small package size in the npm ecosystem could lead to different dynamics than other ecosystems, significantly affecting the dependency migration patterns such as and barriers to migrating to other packages. Future work needs to investigate if a similar approach can also help generate dependency migration suggestions in other ecosystems such as PyPi and Maven.

## **7.7 Chapter Summary**

This chapter presents an approach to extract dependency migration trends in the software ecosystem and suggest alternatives for packages that are in decline. We evaluate our approach in npm, one of the largest and most popular software ecosystems. Our evaluation showed that our approach were accurate at suggesting alternative packages (RQ1). Out of

the 152 dependency migration suggestions generated by our approach, we found that 96% of them are valid alternative package suggestions. We evaluate the usefulness of our approach through a survey with JavaScript developers (RQ2). We found that it helped 52% of the developers to learn about the alternative packages. Also, on a 5-points Likert scale, the developers support having a tool that utilizes our approach to suggest alternative packages with median = 4. Moreover, 67% of the developers will use our suggested alternative packages in their future projects. Finally, we investigated the activities and the motivation of performing our suggested dependency migrations (RQ3). We found that 73% of the pull requests perform dependency migrations alongside other activities such as dependency version update, bug fixing, refactoring. Also, we found that 74% of the dependency migrations are performed to move to well-maintained alternative packages.

Our work outlines some directions for future work. First, in this chapter, we generate suggestions for one-to-one dependency migrations, where one package replaces another. We believe it is valuable for future work to support one-to-many and many-to-many dependency migrations, where one or more packages are replacing one or more packages. Another exciting follow-up work is to propose an automated approach that uses the dependency migration examples to perform the suggested dependency migrations.

# Chapter 8

## Conclusions and Future Work

Software ecosystems have become an essential part of today's software development. The work proposed in this thesis is influenced by the rapid growth of software ecosystems, with which reusing third-party code in the form of packages has become a widespread practice. In this thesis, we focused on helping developers mitigating the challenges in managing open source package dependencies. We started by conducting an empirical study to understand the characteristic of highly-used packages and the factors used to select the packages. Next, we proposed multiple approaches to help developers manage their dependencies, including making an informed decision when updating dependencies, evaluating the packages, identifying packages that could be useful to replace, and finding alternative packages.

### 8.1 Conclusion and Findings

Recent studies have shown that reusing packages from software ecosystems improves productivity, reduces time-to-market, and enhances quality. However, like any solution, using open source dependencies from software ecosystems has challenges such as quality issues, risk of breakage-inducing versions, maintenance issues, and even added complexity.

The research presented in this thesis focuses on helping software developers, who depend on packages from the ecosystem, overcome the challenges in managing their dependencies. More specifically, the presented research provides the following main contributions:

## **Chapter 4: An Empirical Study on the Characteristics of Highly-Selected Packages**

In this chapter, we conducted a mixed qualitative and quantitative study to understand how developers identify and select relevant open source packages. Specifically, we started by surveying 118 JavaScript developers from the npm ecosystem to qualitatively understand the factors that make a package to be highly used within the ecosystem. The survey results showed that JavaScript developers believe that highly-used packages are well-documented, receive a high number of stars on GitHub, have a large number of downloads, and do not suffer from security vulnerabilities. Then, we conducted an experiment to quantitatively validate the developers' perception of the factors that make a highly-used package. Our findings can help improve the package search engines and package recommendation systems.

## **Chapter 5: An Approach to Identify Breaking Updates**

Software dependencies are constantly evolving with newly added features and patches that fix bugs in older versions. However, updating dependencies could introduce new bugs or break backward compatibility. In this chapter, we proposed an approach to detect breakage-inducing versions of third-party dependencies. The underlying rationale is that the test suites of dependent projects can cover more real-world scenarios than the package's tests alone. Thus, our approach leverage the automated test suites of other projects that depend upon the same dependency to test newly released versions. We conjecture that

this crowd-based approach will help detect breakage-inducing versions because it broadens realistic usage scenarios of the packages. To evaluate our conjecture, we performed an empirical study of 391,553 npm packages. We used the dependency network from these packages to identify candidate tests of third-party packages. We mined the history of this dependency network to identify ten breakage-inducing versions. We found that our proposed technique can detect six of the ten studied real-world breakage-inducing versions. Our findings can help developers to make more informed decisions when they update their dependencies.

## **Chapter 6: An Approach to Identify Packages in Decline**

Packages that show a decline in community interest are usually used less over time, become less frequently maintained, and eventually become abandoned. Thus, this chapter proposes a scalable approach that uses the packages' centrality in the ecosystem to identify packages in decline. We evaluated our approach on packages from the npm ecosystem. The results showed that our approach can correctly detect 87% of packages in decline. More crucial, our approach can detect packages in decline on average 18.35 months before their *npm*s scores reflecting the decline. Also, it detects packages in decline on average 16.15 months before a package gets deprecated. In general, the results showed that our approach provides trends that are different from those provided by other metrics such as dependents, downloads, stars, and forks. We implemented a tool that utilizes our approach to help developers avoid packages in decline when reusing packages from the npm ecosystem.

## **Chapter 7: An Approach to Find Alternative Packages**

The decline in community interest may indicate that a better solution is drawing attention in the ecosystem, and developers are migrating to a package that better suits their requirements. This chapter aims to use the crowd knowledge extracted from the software

ecosystem to suggest alternative packages to those in decline. Therefore, we propose an approach that extracts dependency migration patterns and utilizes the centrality trends to suggest package alternatives. Our evaluation shows that our approach suggested valid alternative packages in 96% of the cases. Further, 67% of the developers confirmed that they would use our suggested alternative packages. Also, on a 5-points Likert scale, the developers support having a tool that utilizes our approach to suggest alternative packages with median = 4.

## **8.2 Limitations**

A limitation of our work is that we study data mainly from the npm ecosystem. Thus, we cannot assert whether our results are generalizable to other software ecosystems. Another limitation is that we restrict our study to the packages published on npm ecosystem, while external projects (e.g., published only on Github) can also depend on npm packages. To analyze the dependencies of these external projects, data collection challenges beyond this thesis's scope must be resolved first. It is difficult to collect a comprehensive view of the external projects and their dependency changes over time.

## **8.3 Future Work**

Although this thesis has made many contributions towards facilitating the dependency maintenance activities, many different avenues for future work remain unexplored. We summarize some of the main directions for future work.

### **8.3.1 Replication on Other Software Ecosystems**

Our work focuses on the npm ecosystem. However, other software ecosystems have unique characteristics resulted from the variations in policies, practices, use cases, and

tools. Future research that replicates and adapts our work on other software ecosystems would generalize our approaches and results to help a broader range of developers. To support such future work, through this thesis, we detailed the technical implementations of our experiments and approaches. In addition, we have published our data and scripts to support future replications.

### **8.3.2 Extend Scope of Dependent Projects**

Our proposed approaches rely on observing the dependency network of the packages in the software ecosystem. However, other projects out the ecosystem also depend on the packages in the ecosystem. Extending the scope of the dependency network to include other projects that are not published as packages in the ecosystem can help advance our approaches. In Chapter 5, our approach uses test suites from dependent projects to test newly released package versions. Extending the scope to other projects can increase the number of dependent projects, which improves the accuracy and the applicability of our approach. This improvement will be helpful especially for top-level packages, i.e., packages that are used mainly by applications, not other packages. Likewise, our approaches in Chapters 6 and 7 monitor the changes in the dependency network to calculate the centrality and extract migration patterns. Extending the scope of dependent could improve the quality of the centrality rankings. Also, it will increase the number of extracted migration patterns, which could improve the alternative package suggestions. Future work should explore solving the technical challenging in expanding the scope of the dependency network. Also, it should investigate the impact of extending the scope of the dependency network on the quality of our results.



### **8.3.3 Investigate Why a Package Is in Decline**

The Chapter 6 of this thesis proposes an approach to detect packages in decline, and Chapter 7 proposes an approach to find alternatives for them. An important question left to answer is: why do packages become in decline. Future research should investigate and characterize the reasons that drove packages to become in decline. Furthermore, future work should explore developing an approach to quantitative the reasons for a package to become in decline, which supports developers to make more informed decisions.

### **8.3.4 Predict Future Central Packages**

In Chapter 6, we focus on detecting package in decline as early as possible. However, another important direction is to predict the future central package. Finding future central packages can be used by open source contributors to find packages that will acquire the community's interest. On the other side, these packages can receive the attention needed to boost their evolution as early as possible. Future work should develop an approach to predict the future central packages. Furthermore, it should investigate the impact of such an approach on the practices in the software ecosystems.

### **8.3.5 Expand Dependency Migration Suggestions**

In Chapter 7, our approach generates one-to-one dependency migration suggestions, where one package replaces another. However, more complex dependency migrations could involve replacing multiple packages. Future work should explore supporting one-to-many and many-to-many dependency migrations, where one or more packages are replacing one or more packages.

### **8.3.6 Automate Dependency Migrations**

In this thesis, we propose an approach to suggest dependency migrations, which include real-world examples for open source projects that perform the suggested migrations. However, developers are still required to perform the suggested migrations manually. Future work should explore proposing an automated approach that uses the dependency migration examples to perform the suggested dependency migrations.

# References

- Abdalkareem, R., Mujahid, S., & Shihab, E. (2020). A machine learning approach to improve the detection of CI skip commits. *IEEE Transactions on Software Engineering Journal*, 47(3), 448-463. doi: 10.1109/TSE.2020.2967380
- Abdalkareem, R., Mujahid, S., Shihab, E., & Rilling, J. (2019). Which commits can be CI skipped? *IEEE Transactions on Software Engineering Journal*, 47(3), 448-463. doi: 10.1109/TSE.2019.2897300
- Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., & Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 11th joint meeting on foundations of software engineering* (pp. 385–395). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3106237.3106267
- Abdalkareem, R., Oda, V., Mujahid, S., & Shihab, E. (2020). On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering*, 25(2), 1168–1204.
- Abdellatif, A., Zeng, Y., Elshafei, M., Shihab, E., & Shang, W. (2020). Simplifying the search of npm packages. *Information and Software Technology*, 126, 106365.
- Alrubaye, H., Alshoabi, D., Alomar, E., Mkaouer, M. W., & Ouni, A. (2020). How does library migration impact software quality and comprehension? an empirical study. In S. Ben Sassi, S. Ducasse, & H. Mili (Eds.), *Reuse in emerging software engineering practices* (pp. 245–260). Cham: Springer International Publishing.
- Alrubaye, H., Mkaouer, M. W., & Ouni, A. (2019a). Migrationminer: An automated

- detection tool of third-party java library migration at the method level. In *2019 IEEE International Conference on Software Maintenance and Evolution* (p. 414-417). doi: 10.1109/ICSME.2019.00072
- Alrubaye, H., Mkaouer, M. W., & Ouni, A. (2019b). On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension* (p. 347-357).
- Artzi, S., Dolby, J., Jensen, S. H., Møller, A., & Tip, F. (2011). A framework for automated testing of javascript web applications. In *Proceedings of the 33rd international conference on software engineering* (pp. 571–580). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1985793.1985871
- Barbosa, O., dos Santos, R. P., Alves, C., Werner, C., & Jansen, S. (2013). A systematic mapping study on software ecosystems from a three-dimensional perspective. In *Software ecosystems*. Cheltenham, UK: Edward Elgar Publishing.
- Bavota, G., Linares-Vásquez, M., Bernal-Cárdenas, C. E., Penta, M. D., Oliveto, R., & Shybyanyk, D. (2015). The impact of API change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4), 384-407.
- Begel, A., Bosch, J., & Storey, M. (2013, January). Social networking meets software development: Perspectives from GitHub, MSDN, Stack Exchange, and TopCoder. *IEEE Software*, 30(1), 52-66.
- Benitte, R., & Greif, S. (2021, January). *The state of JavaScript survey*. <https://stateofjs.com/>. ((Accessed on 12/20/2021))
- Berberich, K., Bedathur, S., Weikum, G., & Vazirgiannis, M. (2007). Comparing apples and oranges: Normalized PageRank for evolving graphs. In *Proceedings of the 16th international conference on world wide web* (p. 1145–1146). ACM.

- Bernhardt, J. (2019, July). *Update packages*. <https://github.com/compose-us/todastic/pull/19>. ((Accessed on 09/09/2021))
- Birchler, P. (2020, September). *Replace moment with date-fns*. <https://github.com/google/web-stories-wp/pull/4484>. ((Accessed on 01/30/2021))
- Bogart, C., Kästner, C., Herbsleb, J., & Thung, F. (2016). How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th acm sigsoft international symposium on foundations of software engineering* (pp. 109–120). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2950290.2950325
- Bogart, C., Kästner, C., & Herbsleb, J. (2015, November). When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of the 30th ieee/acm international conference on automated software engineering workshop* (p. 86-89). New York, NY, USA: IEEE. doi: 10.1109/ASEW.2015.21
- Borges, H., Hora, A., & Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *2016 ieee international conference on software maintenance and evolution* (p. 334-344).
- Borges, H., & Valente, M. T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146, 112 - 129.
- Bosch, J. (2010). Architecture challenges for software ecosystems. In *Proceedings of the fourth european conference on software architecture: Companion volume* (p. 93–95). New York, NY, USA: Association for Computing Machinery.
- Brierton, D. (2017, 10). *Restrict version to pre-1.12 as it includes a dep requiring const*. <https://github.com/CoderDojo/cp-users-service/commit/59543709173c3af56baa216318cc4c954639d73b>. ((Accessed on 03/18/2018))

- Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1), 107 - 117. (Proceedings of the Seventh International World Wide Web Conference)
- Bui, N. D. Q., Yu, Y., & Jiang, L. (2019). Sar: Learning cross-language api mappings with little knowledge. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 796–806). New York, NY, USA: Association for Computing Machinery.
- Burrows, D. (2017, October). *What is a package manager?* <https://web.archive.org/web/20171017151526/http://aptitude.alioth.debian.org/doc/en/pr01s02.html>. ((Retrieved 19 December 2018))
- Buse, R. P. L., & Zimmermann, T. (2012). Information needs for software development analytics. In *Proceedings of the 34th international conference on software engineering* (p. 987–996). IEEE Press.
- Businge, J., Serebrenik, A., & van den Brand, M. G. J. (2012, September). Survival of eclipse third-party plug-ins. In *Proceedings of the 28th ieee international conference on software maintenance* (p. 368-377). New York, NY, USA: IEEE. doi: 10.1109/ICSM.2012.6405295
- Businge, J., Serebrenik, A., & van den Brand, M. G. J. (2015, March). Eclipse api usage: The good and the bad. *Software Quality Journal*, 23(1), 107–141. doi: 10.1007/s11219-013-9221-3
- Cabral, G. G., Minku, L. L., Shihab, E., & Mujahid, S. (2019). Class imbalance evolution and verification latency in just-in-time software defect prediction. In *Proceedings of the 41st IEEE/ACM international conference on software engineering* (p. 666-676). doi: 10.1109/ICSE.2019.00076

- Cadariu, M., Bouwers, E., Visser, J., & van Deursen, A. (2015, mar). Tracking known security vulnerabilities in proprietary software systems. In *Proceedings of the 22nd international conference on software analysis, evolution, and reengineering* (pp. 516–519). New York, NY, USA: IEEE. doi: 10.1109/SANER.2015.7081868
- Cadini, F., Zio, E., & Petrescu, C.-A. (2009). Using centrality measures to rank the importance of the components of a complex network infrastructure. In R. Setola & S. Geretshuber (Eds.), *Critical information infrastructure security* (pp. 155–167). Springer Berlin Heidelberg.
- Chatzidimitriou., K. C., Papamichail., M. D., Diamantopoulos., T., Oikonomou., N., & Symeonidis., A. L. (2019). npm packages as ingredients: A recipe-based approach. In *Proceedings of the 14th international conference on software technologies - volume 1: Icsoft*, (p. 544-551). SciTePress.
- Chen, C., Gao, S., & Xing, Z. (2016). Mining analogical libraries in q amp;a discussions – incorporating relational and categorical knowledge into word embedding. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering* (Vol. 1, p. 338-348).
- Chen, C., Xing, Z., Liu, Y., & Xiong, K. O. L. (2021). Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 47(3), 432-447.
- Chen, X., Abdalkareem, R., Mujahid, S., Shihab, E., & Xia, X. (2021, Mar 02). Helping or not helping? why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering Journal*, 26(2), 24. Retrieved from <https://doi.org/10.1007/s10664-020-09904-w> doi: 10.1007/s10664-020-09904-w
- Codecademy. (2021). *Normalization*. <https://www.codecademy.com/articles/normalization>. ((Accessed on 09/02/2021))
- Coelho, J., & Valente, M. T. (2017). Why modern open source projects fail. In

- Proceedings of the 2017 11th joint meeting on foundations of software engineering* (p. 186–196). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3106237.3106246> doi: 10.1145/3106237.3106246
- Coelho, J., Valente, M. T., Milen, L., & Silva, L. L. (2020). Is this GitHub project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, 122, 106274.
- Coelho, J., Valente, M. T., Silva, L. L., & Shihab, E. (2018). Identifying unmaintained projects in GitHub. In *Proceedings of the 12th acm/ieee international symposium on empirical software engineering and measurement*. ACM.
- Cogo, F. R., Oliva, G. A., & Hassan, A. E. (2019). An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, 1-1.
- Cogo, F. R., Oliva, G. A., & Hassan, A. E. (2021). Deprecation of packages and releases in software ecosystems: A case study on npm. *IEEE Transactions on Software Engineering*, 1-1.
- Cohen, J. (1960). A coefficient of agreement for nominal scale. *Educational and Psychological Measurement*, 20, 37–46.
- Cossette, B. E., & Walker, R. J. (2012). Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proceedings of the acm sigsoft 20th international symposium on the foundations of software engineering*. New York, NY, USA: Association for Computing Machinery.
- Costa, D. E., Mujahid, S., Abdalkareem, R., & Shihab, E. (2021). Breaking type-safety in go: An empirical study on the usage of the unsafe package. *IEEE Transactions on Software Engineering Journal*, 1-1. doi: 10.1109/TSE.2021.3057720
- Cruz, A., & Duarte, A. (2018). *npms*. <https://npms.io/about>. ((Accessed on



01/30/2021))

- Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the acm 2012 conference on computer supported cooperative work* (pp. 1277–1286). ACM.
- Daley, N. (2018, November). *Replace moment.js with date-fns by noelledaley*. <https://github.com/hashicorp/vault/pull/5789>. ((Accessed on 01/30/2021))
- DeBill, E. (2021). *Modulecounts*. <http://www.modulecounts.com/>. ((Accessed on 09/25/2021))
- Decan, A., Mens, T., & Claes, M. (2017, February). An empirical comparison of dependency issues in oss packaging ecosystems. In *Proceedings of the 24th international conference on software analysis, evolution and reengineering* (p. 2-12). New York, NY, USA: IEEE. doi: 10.1109/SANER.2017.7884604
- Decan, A., Mens, T., & Constantinou, E. (2018a, September). On the evolution of technical lag in the npm package dependency network. In *Proceedings of the 2018 ieee international conference on software maintenance and evolution* (p. 404-414). New York, NY, USA: IEEE. doi: 10.1109/ICSME.2018.00050
- Decan, A., Mens, T., & Constantinou, E. (2018b). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories* (pp. 181–191). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3196398.3196401
- Decan, A., Mens, T., & Grosjean, P. (2019, February). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1), 381–416. Retrieved from <https://doi.org/10.1007/s10664-017-9589-y> doi: 10.1007/s10664-017-9589-y
- Decan, A., Mens, T., Zerouali, A., & De Roover, C. (2021). Back to the past – analysing

- backporting practices in package dependency networks. *IEEE Transactions on Software Engineering*, 1-1. doi: 10.1109/TSE.2021.3112204
- del Bianco, V., Lavazza, L., Morasca, S., & Taibi, D. (2011). A survey on open source software trustworthiness. *IEEE Software*, 28(5), 67-75.
- de la Mora, F. L., & Nadi, S. (2018). An empirical study of metric-based comparisons of software libraries. In *Proceedings of the 14th international conference on predictive models and data analytics in software engineering* (p. 22–31). New York, NY, USA: Association for Computing Machinery.
- den Besten, M., Amrit, C., Capiluppi, A., & Robles, G. (2020). Collaboration and innovation dynamics in software ecosystems: A technology management research perspective. *IEEE Transactions on Engineering Management*, 1-6.
- Dey, T., Karnauch, A., & Mockus, A. (2021). Representation of developer expertise in open source software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering* (p. 995-1007). doi: 10.1109/ICSE43902.2021.00094
- Dey, T., & Mockus, A. (2018). Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? In *Proceedings of the 14th international conference on predictive models and data analytics in software engineering* (p. 66–69). ACM.
- Dig, D., & Johnson, R. (2006, March). How do apis evolve&quest; a story of refactoring. *Journal of Software Maintenance*, 18(2), 83–107. doi: 10.1002/smr.328
- Dillman, D. A. (2011). *Mail and internet surveys: The tailored design method–2007 update with new internet, visual, and mixed-mode guide*. John Wiley & Sons.
- Dorninger, B., Moser, M., & Pichler, J. (2017). Multi-language re-documentation to support a cobol to java migration project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering* (p. 536-540).

- Erlenhov, L., Neto, F. G. d. O., & Leitner, P. (2020). An empirical study of bots in software development: Characteristics and challenges from a practitioner's perspective. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 445–455). New York, NY, USA: Association for Computing Machinery.
- Fard, A. M., & Mesbah, A. (2017, March). Javascript: The (un)covered parts. In *Proceedings of the 2017 ieee international conference on software testing, verification and validation* (p. 230-240). New York, NY, USA: IEEE. doi: 10.1109/ICST.2017.28
- Fard, A. M., Mesbah, A., & Wohlstadter, E. (2015). Generating fixtures for javascript unit testing (t). In *Proceedings of the 30th ieee/acm international conference on automated software engineering (ase)* (pp. 190–200). Washington, DC, USA: IEEE Computer Society. doi: 10.1109/ASE.2015.26
- Farrell, C. (2019a, April). *Deprecate istanbul-api*. <https://github.com/istanbuljs/istanbuljs/pull/378>. ((Accessed on 06/23/2021))
- Farrell, C. (2019b, March). *Explore deprecation of istanbul-api*. <https://github.com/istanbuljs/istanbuljs/issues/321>. ((Accessed on 06/23/2021))
- Fleiss, J. L., & Cohen, J. (1973). The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and Psychological Measurement*, 33, 613–619.
- Fleurey, F., Breton, E., Baudry, B., Nicolas, A., & Jézéquel, J.-M. (2007). Model-driven engineering for software migration in a large industrial context. In G. Engels, B. Opdyke, D. C. Schmidt, & F. Weil (Eds.), *Model driven engineering languages and systems* (pp. 482–497). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Flyvbjerg, B. (2006). Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2), 219–245.

- Fowler, J., Cohen, L., & Jarvis, P. (2009). *Practical statistics for field biology*. John Wiley & Sons.
- Franch, X., & Carvallo, J. P. (2003). Using quality models in software package selection. *IEEE software*, 20(1), 34–41.
- German, D. M., Adams, B., & Hassan, A. E. (2013, March). The evolution of the R software ecosystem. In *2013 17th european conference on software maintenance and reengineering* (p. 243-252).
- GitHub. (2021a). *GitHub GraphQL API - GitHub docs*. <https://docs.github.com/en/graphql>. ((accessed on 01/27/2021))
- GitHub. (2021b). *GitHub REST API - GitHub docs*. <https://docs.github.com/en/rest>. ((accessed on 01/27/2021))
- Gleich, D. F. (2015, January). PageRank beyond the web. *SIAM Review*, 57(3), 321–363.
- Gokhale, A., Ganapathy, V., & Padmanaban, Y. (2013). Inferring likely mappings between apis. In *2013 35th international conference on software engineering* (p. 82-91). doi: 10.1109/ICSE.2013.6606554
- Golubev, Y., Eliseeva, M., Povarov, N., & Bryksin, T. (2020). A study of potential code borrowing and license violations in java projects on github. In *Proceedings of the 17th international conference on mining software repositories* (p. 54–64). New York, NY, USA: Association for Computing Machinery.
- Gonzalez-Barahona, J. M., Sherwood, P., Robles, G., & Izquierdo, D. (2017). Technical lag in software compilations: Measuring how outdated a software deployment is. In F. Balaguer, R. Di Cosmo, A. Garrido, F. Kon, G. Robles, & S. Zacchiroli (Eds.), *Open source systems: Towards robust practices* (pp. 182–192). Cham: Springer International Publishing.
- Gousios, G., Pinzger, M., & Deursen, A. v. (2014). An exploratory study of the pull-based software development model. In *Proceedings of the 36th international conference*

- on software engineering* (p. 345–355).
- Gousios, G., & Zaidman, A. (2014). A dataset for pull-based development research. In *Proceedings of the 11th working conference on mining software repositories* (p. 368–371). New York, NY, USA: Association for Computing Machinery.
- Greif, S., & Benitte, R. (2019, December). *The state of JavaScript 2019*. <https://2019.stateofjs.com/>. ((Accessed on 12/20/2020))
- Grissom, R. J., & Kim, J. J. (2005). *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers.
- Hacker News. (2015a, December). *Is Sails.js dying?* <https://news.ycombinator.com/item?id=10755557>. ((Accessed on 06/25/2021))
- Hacker News. (2015b, December). *Status of Sails.js*. <https://news.ycombinator.com/item?id=10819583>. ((Accessed on 06/25/2021))
- Hacker News. (2016, December). *Jest: Painless JavaScript testing*. <https://news.ycombinator.com/item?id=13128146>. ((Accessed on 06/24/2021))
- Haefliger, S., von Krogh, G., & Spaeth, S. (2008, January). Code reuse in open source software. *Manage. Sci.*, 54(1), 180–193.
- Haenni, N., Lungu, M., Schwarz, N., & Nierstrasz, O. (2013). Categorizing developer information needs in software ecosystems. In *Proceedings of the 2013 international workshop on ecosystem architectures* (pp. 1–5). ACM.
- Hardcastle, N. (2019, January). *Replace raven with sentry-sdk*. <https://github.com/OpenNeuroOrg/openneuro/pull/1040>. ((Accessed on 09/07/2021))
- Harrell Jr, F. E. (2015). *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer.
- Hata, H., Todo, T., Onoue, S., & Matsumoto, K. (2015, May). Characteristics of sustainable oss projects: A theoretical and empirical study. In *2015 IEEE/ACM 8th International*

- workshop on cooperative and human aspects of software engineering* (p. 15-21).  
IEEE.
- Hauge, O., Osterlie, T., Sorensen, C., & Gereaa, M. (2009, May). An empirical study on selection of open source software - preliminary results. In *2009 icse workshop on emerging trends in free/libre/open source software research and development* (p. 42-47). IEEE.
- He, H., Xu, Y., Ma, Y., Xu, Y., Liang, G., & Zhou, M. (2021). A multi-metric ranking approach for library migration recommendations. In *2021 ieee international conference on software analysis, evolution and reengineering (saner)* (p. 72-83).
- Hong, J. B., Kim, D. S., & Haqiq, A. (2014). What vulnerability do we need to patch first? In *2014 44th annual ieee/ifip international conference on dependable systems and networks* (p. 684-689).
- Hoyos, J., Abdalkareem, R., Mujahid, S., Shihab, E., & Bedoya, A. E. (2021, Feb 03). On the removal of feature toggles. *Empirical Software Engineering Journal*, 26(2), 15. Retrieved from <https://doi.org/10.1007/s10664-020-09902-y> doi: 10.1007/s10664-020-09902-y
- Iasonos, A., Schrag, D., Raj, G. V., & Panageas, K. S. (2008). How to build and interpret a nomogram for cancer prognosis. *Journal of clinical oncology*, 26(8), 1364–1370.
- Inc., F. (2021, March). *react on the npm website*. <https://www.npmjs.com/package/react>. ((Accessed on 09/12/2021))
- Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., & Kusumoto, S. (2005). Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3), 213-225.
- Jadhav, A. S., & Sonar, R. M. (2009). Evaluating and selecting software packages: A review. *Information and Software Technology*, 51(3), 555-563.
- James, F. E. (1968). Monthly moving averages—an effective investment tool? *Journal of*

- Financial and Quantitative Analysis*, 3(3), 315–326. doi: 10.2307/2329816
- Jest Blog. (2016a, December). *2016 in Jest*. <https://jestjs.io/blog/2016/12/15/2016-in-jest>. ((Accessed on 06/24/2021))
- Jest Blog. (2016b, March). *JavaScript unit testing performance*. <https://jestjs.io/blog/2016/03/11/javascript-unit-testing-performance>. ((Accessed on 06/24/2021))
- John, W., Creswell, P. C., & CLARK, V. (2000). Designing and conducting mixed methods research..
- Johnson-Pint, M. (2020, September). *Project status - moment.js*. <https://momentjs.com/docs/#/-project-status/>. ((Accessed on 12/30/2020))
- jQuery Foundation, T. (2019). *jquery javascript library*. <https://github.com/jquery/jquery>. ((Accessed on 01/20/2019))
- Judge, G. G., Griffiths, W. E., Hill, R. C., Lutkepohl, H., & Lee, T.-C. (1985). *The theory and practice of econometrics*. Wiley.
- Kabinna, S., Bezemer, C.-P., Shang, W., & Hassan, A. E. (2016). Logging library migrations: A case study for the apache software foundation projects. In *2016 IEEE/ACM 13th working conference on mining software repositories (msr)* (p. 154-164).
- Kapur, P., Cossette, B., & Walker, R. J. (2010). Refactoring references for library migration. In *Proceedings of the ACM international conference on object oriented programming systems languages and applications* (p. 726–738). New York, NY, USA: Association for Computing Machinery.
- Kashcha, A. (2017). *npm packages sorted by pagerank*. <http://anvaka.github.io/npmrank/online/>. ((Accessed on 11/24/2017))
- Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika*, 30(1/2), 81–93.
- Khondhu, J., Capiluppi, A., & Stol, K.-J. (2013). Is it all lost? a study of inactive open source projects. In E. Petrinja, G. Succi, N. El Ioini, & A. Sillitti (Eds.), *Open source*

- software: Quality verification* (pp. 61–79). Springer Berlin Heidelberg.
- Kikas, R., Gousios, G., Dumas, M., & Pfahl, D. (2017). Structure and evolution of package dependency networks. In *Proceedings of the 14th international conference on mining software repositories* (pp. 102–112). IEEE Press.
- Kim, M., Nam, J., Yeon, J., Choi, S., & Kim, S. (2015). Remi: Defect prediction for efficient api testing. In *Proceedings of the 10th joint meeting on foundations of software engineering* (pp. 990–993). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2786805.2804429
- Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2017). *Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration*. doi: 10.1007/s10664-017-9521-5
- Larios Vargas, E., Aniche, M., Treude, C., Bruntink, M., & Gousios, G. (2020). Selecting third-party libraries: The practitioners’ perspective. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 245–256). ACM.
- Lee, D., Rajbahadur, G. K., Lin, D., Sayagh, M., Bezemer, C.-P., & Hassan, A. E. (2020). An empirical study of the characteristics of popular minecraft mods. *Empirical Software Engineering*, 25(5), 3396–3429.
- Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4), 485-496.
- Li, H., Shang, W., Zou, Y., & Hassan, A. E. (2017). Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4), 1831–1865.
- Lindesay, F. (2016, June). *Jade node template engine - npm*. <https://www.npmjs.com/package/jade>. ((Accessed on 02/01/2021))



- Lungu, M., Lanza, M., Gîrba, T., & Robbes, R. (2010). The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4), 264 - 275. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0167642309001221> (Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008)) doi: <https://doi.org/10.1016/j.scico.2009.09.004>
- Ma, W., Chen, L., Zhang, X., Feng, Y., Xu, Z., Chen, Z., . . . Xu, B. (2020). Impact analysis of cross-project bugs on software ecosystems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering* (pp. 100–111).
- Ma, Y., Mockus, A., Zaretzki, R., Bichescu, B., & Bradley, R. (2020). A methodology for analyzing uptake of software technologies among developers. *IEEE Transactions on Software Engineering*, 1-1. doi: 10.1109/TSE.2020.2993758
- Maharani, W., Adiwijaya, & Gozali, A. A. (2014). Degree centrality and eigenvector centrality in Twitter. In *2014 8th International Conference on Telecommunication Systems Services and Applications* (p. 1-5). IEEE.
- Mahmoud Alfadel, E. S., Diego Elias Costa. (2021). Empirical analysis of security vulnerabilities in Python packages. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (to appear)*.
- Manaskasemsak, B., & Rungsawang, A. (2005). An efficient partition-based parallel PageRank algorithm. In *11th International Conference on Parallel and Distributed Systems* (Vol. 1, p. 257-263 Vol. 1).
- Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 50–60.
- McHugh, M. (2012, 10). Interrater reliability: The kappa statistic. *Biochemia medica*, 22, 276-82.
- Meloca, R., Pinto, G., Baiser, L., Mattos, M., Polato, I., Wiese, I. S., & German, D. M.

- (2018). Understanding the usage, impact, and adoption of non-osi approved licenses. In *Proceedings of the 15th international conference on mining software repositories* (pp. 270–280). Association for Computing Machinery.
- Mezzetti, G., Møller, A., & Torp, M. T. (2018). Type regression testing to detect breaking changes in node.js libraries. In T. Millstein (Ed.), *Proceedings of the 32nd european conference on object-oriented programming* (Vol. 109, pp. 7:1–7:24). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.ECOOP.2018.7
- Midi, H., Sarkar, S. K., & Rana, S. (2010). Collinearity diagnostics of binary logistic regression model. *Journal of Interdisciplinary Mathematics*, 13(3), 253–267.
- Mirhosseini, S., & Parnin, C. (2017, Oct). Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd ieee/acm international conference on automated software engineering* (pp. 84–94). Piscataway, NJ, USA: IEEE Press. doi: 10.1109/ASE.2017.8115621
- Mirshokraie, S., Mesbah, A., & Pattabiraman, K. (2015, April). Jseft: Automated javascript unit test generation. In *Proceedings of the 8th ieee international conference on software testing, verification and validation* (p. 1-10). New York, NY, USA: IEEE. doi: 10.1109/ICST.2015.7102595
- Møller, A., & Torp, M. T. (2019). Model-based testing of breaking changes in node.js libraries. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 409–419). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3338906.3338940> doi: 10.1145/3338906.3338940
- Mostafa, S., Rodriguez, R., & Wang, X. (2017, may). A Study on Behavioral Backward

- Incompatibility Bugs in Java Software Libraries. In *Proceedings of the 39th international conference on software engineering companion* (pp. 127–129). New York, NY, USA: IEEE. doi: 10.1109/ICSE-C.2017.101
- Mujahid, S., Abdalkareem, R., Shihab, E., & McIntosh, S. (2019, January). *Dataset: Using others' tests to avoid breaking updates*. Retrieved from <https://doi.org/10.5281/zenodo.2549129> doi: 10.5281/zenodo.2549129
- Mujahid, S., Abdalkareem, R., Shihab, E., & McIntosh, S. (2020). Using others' tests to identify breaking updates. In *Proceedings of the 17th international conference on mining software repositories* (p. 466–476). New York, NY, USA: Association for Computing Machinery.
- Mujahid, S., Costa, D. E., Abdalkareem, R., & Shihab, E. (2021a, January). *Replication package: Towards using package centrality trend to identify packages in decline*. Zenodo. Retrieved from <https://doi.org/10.5281/zenodo.5003442> doi: 10.5281/zenodo.5003442
- Mujahid, S., Costa, D. E., Abdalkareem, R., & Shihab, E. (2021b, October). *Replication package: Where to go now? finding alternatives for declining packages in the npm ecosystem*. Zenodo. Retrieved from <https://doi.org/10.5281/zenodo.5548231> doi: 10.5281/zenodo.5548231
- Mujahid, S., Costa, D. E., Abdalkareem, R., Shihab, E., Saied, M. A., & Adams, B. (2021, October). Towards using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Management Journal*, 15.
- Nam, J., & Kim, S. (2015). Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM international conference on automated software engineering (ase)* (p. 452–463). IEEE.
- Nanavati, S. (2020, July). *new audit: add large-javascript-libraries audit*. <https://github.com/GoogleChrome/lighthouse/pull/11096>. ((Accessed on

01/30/2021))

- Nguyen, A. T., Tu, Z., & Nguyen, T. N. (2016). Do contexts help in phrase-based, statistical source code migration? In *2016 IEEE International Conference on Software Maintenance and Evolution* (p. 155-165).
- Nguyen, P. T., Di Rocco, J., Di Ruscio, D., & Di Penta, M. (2020). Crossrec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software, 161*, 110460.
- NIST/SEMATECH. (2012, April). *e-handbook of statistical methods: Linear least squares regression*. <https://www.itl.nist.gov/div898/handbook/pmd/section1/pmd141.htm>. ((Accessed on 01/30/2021))
- Nita, M., & Notkin, D. (2010). Using twinning to adapt programs to alternative APIs. In *2010 ACM/IEEE 32nd International Conference on Software Engineering* (Vol. 1, p. 205-214). doi: 10.1145/1806799.1806832
- npm. (2017a). <https://www.npmjs.com/>. ((Accessed on 11/24/2017))
- npm. (2017b). *registry — npm docs*. <https://docs.npmjs.com/cli/v6/using-npm/registry>. ((accessed on 12/24/2020))
- npm Docs. (2021). *Install packages*. <https://docs.npmjs.com/cli/v6/commands/npm-install>. ((Accessed on 01/21/2021))
- npm Documentation. (2019, January). *npm-registry — npm documentation*. <https://docs.npmjs.com/misc/registry>. ((Accessed on January 21, 2019))
- npm Documentation. (2018). *How to use semantic versioning*. Retrieved 2018-02-05, from <https://docs.npmjs.com/getting-started/semantic-versioning>
- npm Documentations. (2018). *How to publish and update a package*. Retrieved 2018-02-05, from <https://docs.npmjs.com/getting-started/publishing-npm-packages>

- npm. (2016). *About npm*. <https://npm.io/about>. ((Accessed on 19 April 2018))
- Oppenheim, A. N. (1992). *Questionnaire design, interviewing and attitude measurement*. Pinter Publishers.
- Ouni, A., Kula, R. G., Kessentini, M., Ishio, T., German, D. M., & Inoue, K. (2017). Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 83, 55-75.
- Page, L., Brin, S., Motwani, R., & Winograd, T. (1999, November). *The PageRank citation ranking: Bringing order to the web*. (Technical Report No. 1999-66). Stanford, CA, USA: Stanford InfoLab. (Previous number = SIDL-WP-1999-0120)
- Pakers, O. (2019, December). *Replace bcrypt with bcryptjs*. <https://github.com/keystonejs/keystone/pull/2053>. ((Accessed on 09/06/2021))
- Pano, A., Graziotin, D., & Abrahamsson, P. (2018, Dec 01). Factors and actors leading to the adoption of a JavaScript framework. *Empirical Software Engineering*, 23(6), 3503–3534.
- Papamichail, M., Diamantopoulos, T., & Symeonidis, A. (2016). User-perceived source code quality estimation based on static analysis metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security* (p. 100-107).
- Parnas, D. L. (1994). Software aging. In *Proceedings of 16th international conference on software engineering* (pp. 279–287).
- Parsa, P. (2018, September). *Use terser-webpack-plugin*. <https://github.com/nuxt/nuxt.js/pull/3928>. ((Accessed on 09/07/2021))
- Patra, J., Dixit, P. N., & Pradel, M. (2018). Conflictjs: Finding and understanding conflicts between JavaScript libraries. In *Proceedings of the 40th international conference on software engineering* (p. 741–751). ACM.
- Preston-Werner, T. (2020, 1). *Semantic versioning 2.0.0*. <https://semver.org/>.

((Accessed on 01/25/2020))

- Qiu, H. S., Li, Y. L., Padala, S., Sarma, A., & Vasilescu, B. (2019). The signals that potential contributors look for when choosing open-source projects. In *Acm conference on computer-supported cooperative work and social computing*. ACM.
- Qiu, S., Kula, R. G., & Inoue, K. (2018). Understanding popularity growth of packages in JavaScript package ecosystem. In *2018 IEEE International Conference on Big Data, Cloud Computing, Data Science & Engineering (BCD)* (pp. 55–60).
- Quixada, L. (2017, October). *Dropped dependency isomorphic-fetch in favor of cross-fetch (react native compatible)*. <https://github.com/apollographql/apollo-fetch/pull/71>. ((Accessed on 08/31/2021))
- Raemaekers, S., van Deursen, A., & Visser, J. (2014, Sep.). Semantic versioning versus breaking changes: A study of the maven repository. In *Proceedings of the 14th international working conference on source code analysis and manipulation* (p. 215-224). New York, NY, USA: IEEE. doi: 10.1109/SCAM.2014.30
- Raemaekers, S., van Deursen, A., & Visser, J. (2017). Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129, 140 - 158. doi: 10.1016/j.jss.2016.04.008
- Rea, L. M., & Parker, R. A. (2014). *Designing and conducting survey research: A comprehensive guide*. John Wiley & Sons.
- Rodríguez-Baquero, D., & Linares-Vásquez, M. (2018). Mutode: Generic javascript and node.js mutation testing tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 372–375). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3213846.3229504
- Saied, M. A., Ouni, A., Sahraoui, H., Kula, R. G., Inoue, K., & Lo, D. (2018). Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software*, 145, 164-179.

- Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4), 557–572.
- Sebestyen, I. (2009, April). *Ecma international finalises major revision of ecmascript*. [http://www.ecma-international.org/news/PressReleases/PR\\_Ecma\\_finalises\\_major\\_revision\\_of\\_ECMAScript.htm](http://www.ecma-international.org/news/PressReleases/PR_Ecma_finalises_major_revision_of_ECMAScript.htm). ((Accessed on 01/22/2019))
- Semeteys, R. (2008, 05/2008). Method for qualification and selection of open source software [Articles]. *Open Source Business Resource*.
- Sheoran, J., Blincoe, K., Kalliamvakou, E., Damian, D., & Ell, J. (2014). Understanding "watchers" on GitHub. In *Proceedings of the 11th working conference on mining software repositories* (pp. 336–339). ACM.
- Silbermann, S. (2020, February). *Switch to cross-fetch*. <https://github.com/mui-org/material-ui/pull/19644>. ((Accessed on 09/07/2021))
- Slagle, A. (2016, April). *Switched to dompurify for html sanitization*. <https://github.com/NYPL-Simplified/opds-web-client/pull/115>. ((Accessed on 09/07/2021))
- Smith, E., Loftin, R., Murphy-Hill, E., Bird, C., & Zimmermann, T. (2013). Improving Developer Participation Rates in Surveys. In *2013 6th international workshop on cooperative and human aspects of software engineering (chase)* (pp. 89–92). IEEE. <https://www.cs.umd.edu/~tedks/papers/2013-CHASE-participation.pdf>.
- Snyk. (2021). *Snyk — developer security — develop fast. stay secure*. <https://snyk.io/>. ((accessed on 01/31/2021))
- StackOverflow. (2017). *node.js - how to find search/find npm packages - stack overflow*. <https://stackoverflow.com/questions/10568512/how-to-find-search-find-npm-packages>. ((Accessed on 11/24/2017))

- Stack overflow developer survey 2018*. (2018, March). <https://insights.stackoverflow.com/survey/2018/>. ((Accessed on 10/26/2018))
- Stergiopoulos, G., Theocharidou, M., Kotzanikolaou, P., & Gritzalis, D. (2015). Using centrality measures in dependency risk graphs for efficient risk mitigation. In M. Rice & S. Shenoi (Eds.), *Critical infrastructure protection ix* (pp. 299–314). Springer International Publishing.
- Taneja, K., Zhang, Y., & Xie, T. (2010). Moda: Automated test generation for database applications via mock objects. In *Proceedings of the ieee/acm international conference on automated software engineering* (pp. 289–292). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1858996.1859053
- Team, S. E. (2019, October). *Top open source licenses and legal risk*. <https://www.synopsys.com/blogs/software-security/top-open-source-licenses/>. ((accessed on 12/20/2020))
- Temple, C. (2017). *npm discover · see what everyone else is using*. <http://www.npmdiscover.com/>. ((Accessed on 11/24/2017))
- Teyton, C., Falleri, J.-R., & Blanc, X. (2012). Mining library migration graphs. In *2012 19th working conference on reverse engineering* (p. 289-298). doi: 10.1109/WCRE.2012.38
- Teyton, C., Falleri, J.-R., Palyart, M., & Blanc, X. (2014). A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11), 1030-1052.
- Thung, F., Lo, D., & Lawall, J. (2013). Automated library recommendation. In *2013 20th working conference on reverse engineering* (p. 182-191).
- Tian, Y., Nagappan, M., Lo, D., & Hassan, A. E. (2015). What are the characteristics of high-rated apps? a case study on free android applications. In *2015 ieee international conference on software maintenance and evolution (icsme)* (p. 301-310).



- Trockman, A., Zhou, S., Kästner, C., & Vasilescu, B. (2018). Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th international conference on software engineering* (pp. 511–522).
- Valiev, M., Vasilescu, B., & Herbsleb, J. (2018). Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 644–655). ACM.
- van den Berk, I., Jansen, S., & Luinenburg, L. (2010). Software ecosystems: A software ecosystem strategy assessment model. In *Proceedings of the fourth european conference on software architecture: Companion volume* (p. 127–134). New York, NY, USA: Association for Computing Machinery.
- Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., & Filkov, V. (2015). Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 10th joint meeting on foundations of software engineering* (pp. 805–816). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2786805.2786850
- Verhaeghe, B., Etien, A., Anquetil, N., Seriai, A., Deruelle, L., Ducasse, S., & Derras, M. (2019). Gui migration using mde from gwt to angular 6: An industrial case. In *2019 ieee 26th international conference on software analysis, evolution and reengineering (saner)* (p. 579-583).
- Wang, J., Guo, W., & Szeto, K. Y. (2017). Optimization of financial network stability by genetic algorithm. In *Proceedings of the international conference on web intelligence* (p. 517–524). ACM.
- Wang, Y., Wen, M., Liu, Y., Wang, Y., Li, Z., Wang, C., ... Zhu, Z. (2020). Watchman: monitoring dependency conflicts for Python library ecosystem. In *Proceedings of the acm/ieee 42nd international conference on software engineering* (pp. 125–135).
- Wasike, S. N. (2010). Selection process of open source software component..

- Wasserman, S., & Faust, K. (1994). *Social network analysis: methods and applications*. Cambridge University Press.
- Waterloo, B. (2020, October). *Switch vscode-azureappservice to dayjs*. <https://github.com/Microsoft/vscode-azuretools/pull/808>. ((Accessed on 09/07/2021))
- Wittern, E., Suter, P., & Rajagopalan, S. (2016). A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th international conference on mining software repositories* (pp. 351–361). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2901739.2901743
- Xavier, L., Brito, A., Hora, A., & Valente, M. T. (2017, Feb). Historical and impact analysis of api breaking changes: A large-scale study. In *Proceedings of the 24th international conference on software analysis, evolution and reengineering* (p. 138-147). New York, NY, USA: IEEE. doi: 10.1109/SANER.2017.7884616
- Xu, B., An, L., Thung, F., Khomh, F., & Lo, D. (2019, Sep 05). Why reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*.
- Yamashita, K., Kamei, Y., McIntosh, S., Hassan, A. E., & Ubayashi, N. (2016). Magnet or sticky? measuring project characteristics from the perspective of developer attraction and retention. *Journal of Information Processing*, 24(2), 339-348.
- Yan, M., Xia, X., Shihab, E., Lo, D., Yin, J., & Yang, X. (2019). Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering*, 45(12), 1211-1229.
- Yu, E., & Deng, S. (2011). Understanding software ecosystems: A strategic modeling approach. In *Proceedings of the third international workshop on software ecosystems* (pp. 65–76).
- Zakas, N. C. (2018). *About - eslint - pluggable javascript linter*. <https://eslint>

[.org/docs/about/](https://www.javascript.org/docs/about/). JS Foundation. ((Accessed on 19 April 2018))

- Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., & Ihara, A. (2018). Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 559–563).
- Zerouali, A., Constantinou, E., Mens, T., Robles, G., & González-Barahona, J. (2018). An empirical analysis of technical lag in npm package dependencies. In R. Capilla, B. Gallina, & C. Cetina (Eds.), *New opportunities for software reuse* (pp. 95–110). Cham: Springer International Publishing. doi: 10.1007/978-3-319-90421-4\_6
- Zerouali, A., & Mens, T. (2017). Analyzing the evolution of testing library usage in open source Java projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering* (p. 417-421).
- Zerouali, A., Mens, T., Robles, G., & Gonzalez-Barahona, J. M. (2019). 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 (SANER)* (pp. 589–593).
- Zheng, W., Zhang, Q., & Lyu, M. (2011). Cross-library API recommendation using web search engines. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (pp. 480–483).
- Zhong, H., & Mei, H. (2018, December). An empirical study on api usages. *IEEE Transactions on Software Engineering*, *45*(4), 319-334. doi: 10.1109/TSE.2017.2782280
- Zhou, S., Vasilescu, B., & Kästner, C. (2019). What the fork: A study of inefficient and efficient forking practices in social coding. In (p. 350–361). ACM.
- Zhu, J., & Wei, J. (2019). An empirical study of multiple names and email addresses in oss version control repositories. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories* (p. 409-420).

Zhu, J., Zhou, M., & Mockus, A. (2014). Patterns of folder use and project popularity: A case study of GitHub repositories. In *Proceedings of the 8th acm/ieee international symposium on empirical software engineering and measurement*. ACM.