Insights into Dependency Maintenance Trends in the Maven Ecosystem

Barisha Chowdhury[1]   Md Fazle Rabbi[2]   S. M. Mahedy Hasan[1]   Minhaz F. Zibran[2]

[1]Department of Computer Science, Rajshahi University of Engineering & Technology, Rajshahi, Bangladesh
[2]Department of Computer Science, Idaho State University, Pocatello, ID, United States
nitub81@gmail.com, mdfazlerabbi@isu.edu, mahedy@cse.ruet.ac.bd, zibran@isu.edu

Abstract—**As modern software development increasingly relies on reusable libraries and components, managing dependencies has become critical for ensuring software stability and security. However, challenges such as outdated dependencies, missed releases, and the complexity of interdependent libraries can significantly impact project maintenance. In this paper, we present a quantitative analysis of the Neo4j dataset using the Goblin framework to uncover patterns of freshness in projects with different numbers of dependencies. Our analysis reveals that releases with fewer dependencies have a higher number of missed releases. Additionally, our study shows that the dependencies in the latest releases have positive freshness scores, indicating better software management efficacy. These results can encourage better management practices and contribute to the overall health of software ecosystems.**

Index Terms—**Dependency management, freshness patterns, Maven, software stability, missed releases**

I. Introduction

The growing complexity of software systems has led to more use of external dependencies. These dependencies help developers build software quickly, reduce redundancy, and enhance functionality. They also promote standardization and improve interoperability across different software systems [1] [2]. Repositories such as Maven Central play an important role in the software development ecosystem by hosting millions of libraries and their releases [3]. Developers can access these libraries to speed up innovation and reduce development efforts. The variety of libraries available allows developers to address domain-specific challenges efficiently [4].

However, using so many dependencies creates problems, such as outdated libraries, missed updates, and compatibility issues between evolving software components. Keeping track of these dependencies and managing them can be difficult. A key but often overlooked factor in dependency management is *freshness* [5]. Freshness measures how current a release is by looking at how many newer releases are available and how much time has passed since the latest release. This metric is essential for managing dependencies effectively.

Despite the importance of these metrics, there is a lack of studies that explore how freshness varies across different types of dependencies. This gap leaves project maintainers without enough insights into the health of their dependencies, which makes it harder for them to make informed decisions about updates and version management.

In our study, we investigate how the dependency structures in the Maven ecosystem affect the maintenance and release management of software projects. Specifically, we want to understand if projects with a larger number of dependencies face more challenges in keeping their dependencies up to date and experiencing longer periods of outdated dependencies. We also want to explore how much of the dependencies in the latest releases are outdated, which can highlight the difficulties of maintaining up-to-date software in complex dependency environments. Our main goal is to uncover freshness patterns across a few different areas and address the following research questions (RQs):

RQ1: Do projects with a large number of dependencies tend to have a higher "outdated time" or missed releases compared to those with fewer dependencies?

– The findings from this question can provide valuable insights into how the number of dependencies affects project maintenance. By understanding the relationship between dependency counts and outdated dependencies, developers can optimize update processes, reduce delays, and ensure that dependencies remain up-to-date. This can lead to more efficient and stable project management.

RQ2: To what extent are the dependencies in the latest software releases outdated?

– If dependencies in recent releases are found to be outdated, it may indicate gaps in update practices or challenges in maintaining compatibility with newer versions. This could have implications for software stability and security. Understanding these trends can help developers prioritize critical updates, improve dependency management strategies, and make more informed decisions when choosing or maintaining dependencies.

To address the aforementioned RQs, we use the Goblin Miner tool [6] and the Maven Central Neo4j dataset [7]. By using this dataset and tool, we aim to highlight the challenges and trends in managing dependencies within the Maven Central ecosystem. A comprehensive replication package [8], including datasets, analysis scripts, and documentation, has been provided to ensure the reproducibility of this study's findings.

II. Dataset

For this study, we use the publicly available Neo4j Maven Central dependency graph dataset [7]. We work with the "with_metrics_goblin_maven_30_08_24.dump," dated August 30, 2024. The database contains over 15 million nodes, including 658,078 libraries (artifacts) and 14,459,139 releases. It also

includes 134 million edges, with 119,660,406 dependencies and 14,459,139 versioning relationships. The dataset adopts a dependency graph model with two main types of nodes: Artifacts (libraries) and Releases (specific versions of artifacts). Edges in the graph represent release-to-artifact (R→A) and artifact-to-release (A→R) relationships, showing dependency and versioning structures. The dataset also includes additional metadata, such as version ranges, release timestamps, and scopes (e.g., compile, test), which allow for detailed analysis.

The graph structure allows us to compute various metrics like freshness, release rhythm, and vulnerability exposure (CVE data) on-demand using the Goblin framework's Weaver component. This flexibility makes the dataset suitable for exploring key questions about dependency freshness, release patterns, and ecosystem health.

To ensure the feasibility of the study, we analyze a subset of the data, focusing on 100,000 libraries and 1,000,000 dependencies. This allows us to examine dependency management practices while maintaining computational efficiency. This dataset forms the basis for investigating trends and challenges in software dependency management within large ecosystems.

III. Analysis and Results

A. Dependency Counts and Freshness

1) Methodology: We use the *dependency* relationship between releases and artifacts to determine the number of dependencies a release possesses. To assess freshness, we extract different freshness scores using the *AddedValue* edge. The dataset contains 119,660,406 dependencies, but for computational convenience, we analyze a subset of 1,000,000 dependencies ($\approx$ 0.84% of the total). This subset of dependencies comes from 107,916 releases, with the minimum number of dependencies for a single release being 1 and the maximum being 417. By selecting this smaller subset, we balance computational efficiency with the need to capture a broad range of dependency management practices. Although the subset represents only 0.84% of the total dependencies, it still includes data from over 107,000 releases, which is large enough to reflect diverse patterns and trends within the dataset. Dependencies in software ecosystems often follow predictable patterns [9], making this subset large enough for meaningful analysis.

2) Findings: Figure 1 represents the Kernel Density Estimation (KDE) [10] plot for the distribution of dependencies. We observe that most of the data is concentrated around the lower values on the x-axis, near 0 dependencies. This indicates that most releases have very few dependencies. The long tail on the right shows that a small number of releases have very high numbers of dependencies, but these are outliers and occur less frequently. The peak density is slightly above 0.10, representing around 10% of the data.

Figure 2 shows the relationship between dependency count and the number of missed releases across projects. We observe that most projects have fewer than 50 dependencies. These projects have a wide range of missed releases, from zero to over 4000. We notice that projects with high numbers
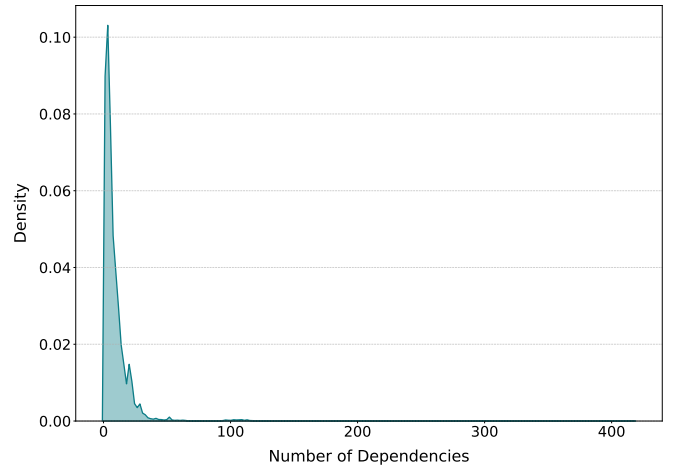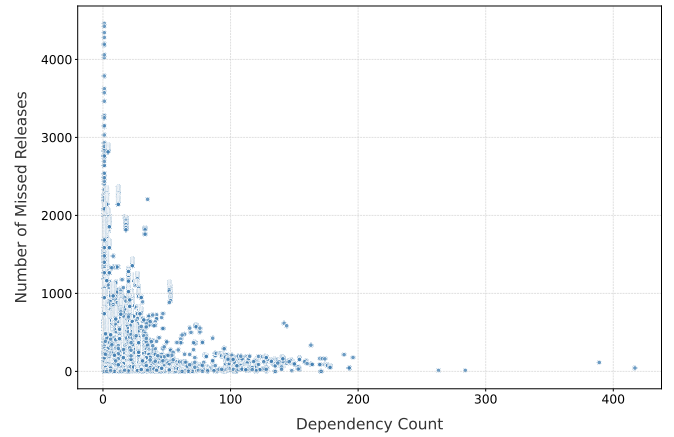

Fig. 1. Distribution of Dependencies

Fig. 2. Dependency Count Vs Number of Missed Releases

of missed releases, such as those exceeding 1000, often have low to moderate dependency counts. This suggests that missed releases are not always caused by a high number of dependencies.

We find that projects with fewer (less than 50) dependencies, which make up nearly 80% of the dataset, have a lot of variation in missed releases. These projects often miss around 1500 releases on average. Smaller projects often lack the resources or maintenance practices needed to update dependencies on time. Projects with moderate (50 to 200) dependencies tend to show more consistent updates, averaging around 500 missed releases. High-dependency (over 200) projects typically miss fewer than 100 releases. This shows that, despite their complexity, projects with more dependencies manage updates more effectively.

Further supporting these findings, Figure 3 shows the outdated times of dependencies for projects with different dependency counts. We find that low-dependency projects have outdated times ranging up to 17.5 years, with an average of about 6 years. Projects with moderate dependency counts show better control, with an average outdated time of about 2.5 years. High-dependency projects have the lowest outdated times, with most staying under 2 years, having some
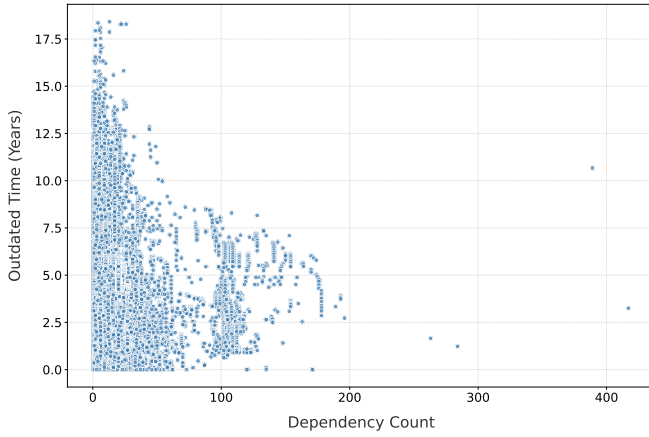
Fig. 3. Dependency Count Vs Outdated Time (Years)

Fig. 4. Distribution of Outdated Time (Years) Across Dependencies

Fig. 5. Distribution of Missed Releases Across Dependencies

exceptions. These projects benefit from regular release cycles, automated dependency tracking, and proactive updates.

The results show an inverse relationship between dependency counts and maintenance challenges. Smaller projects tend to struggle with more missed releases and longer outdated times. Larger projects, however, benefit from structured and proactive dependency management.

B. Dependency of Latest Releases and Freshness

1) Methodology: To identify the latest releases for each artifact, we use the *relationship_AR* edge, which connects artifacts to their releases. We analyze the dependencies of these latest releases and their freshness using the dependency edge along with the *AddedValue* edge. Out of the 658,078 artifacts in the dataset, we select a subset of 100,000 libraries ($\approx$ 15.2%) to ensure computational feasibility while maintaining statistical validity, as recommended in prior studies [11]. The selected libraries contain 742,492 dependencies. However, freshness data is unavailable for 32,066 dependencies, about 4.3% of the total. As a result, we proceed with the remaining 710,426 dependencies (95.7%).

2) Findings: Figure 4 shows the distribution of outdated times across the dependencies of the latest releases. We see a sharp concentration near zero, which indicates that most dependencies are up-to-date. However, a small subset of dependencies has significantly higher outdated times, with some exceeding multiple years. The mean outdated time for dependencies is 2.5 years. This means that while many dependencies are current, some experience delays in updates.

Figure 5 shows a similar distribution for missed releases across dependencies. The density is concentrated near zero, which suggests that most dependencies have few or no missed releases. However, there are some dependencies with a large number of missed releases, but these are relatively rare.

The results suggest that dependencies in the latest releases are generally well-maintained, with minimal missed releases and short outdated times for most cases. However, some dependencies have long outdated times and numerous missed releases, indicating th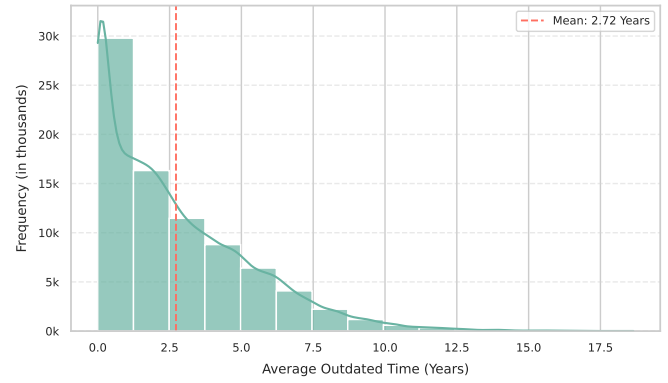at updates have been neglected in these cases. Whether a mean outdated time of 2.5 years is problematic depends on the project's needs and how it manages dependencies. In fast-evolving domains like web frameworks and cloud applications, updates are needed every few months to keep up with changing technologies and security requirements [12]. In these cases, a 2.5-year delay could lead to significant technical debt. For more stable or legacy systems, however, this delay might be acceptable, as older dependencies can still function effectively [13] [14].

IV. THREATS TO VALIDITY

Our methodology provides a quantitative analysis of dependency management trends based on a large-scale dataset. However, we must acknowledge several limitations to help contextualize our findings and their applicability.

We rely on a subset of the Maven Central dataset, consisting of 100,000 libraries and 1,000,000 dependencies. This raises concerns about how well our results apply to the broader dataset. While we chose this subset for computational feasibility, it may not capture patterns found in the larger dataset with millions of artifacts and releases. This limitation could overlook trends in less commonly used libraries or niche areas within the ecosystem. Additionally, we focus only on Maven Central, which, while widely used, is just one software repository. Other ecosystems, such as PyPI or npm, may have different dependency structures and practices, which limits the generalizability of our findings.

Another limitation comes from the simplicity of the metrics we use. Metrics like outdatedness and missed releases provide valuable insights but do not capture more complex aspects of dependency management. For instance, indirect dependencies, scope-specific analyses (e.g., test vs. compile dependencies), and varying update policies could offer a deeper understanding of the challenges developers face.

Additionally, our results are based on data-driven analysis and lack validation from case studies or developer interviews. Qualitative approaches could provide richer insights and confirm the observed trends, especially correlations like those between dependency counts and missed releases. Another potential bias arises from how we select libraries for the analysis. We choose libraries based on feasibility criteria, which may skew the sample toward well-maintained or frequently updated projects. Lastly, we do not account for external factors that may influence dependency management and release practices. Community size, project funding, and developer engagement can all affect how well dependencies are maintained and updated.

V. Related Works

Many studies in the past involved the investigation of bug patterns [15]–[17], vulnerabilities [18], [19], code smells [20]–[22], code quality [20], [23], human aspects [24]–[28] of software development and maintenance as well as comparison of methods/tools [29]–[32] for measuring such aspects.

Software dependency management has become a key focus in software engineering, especially in large ecosystems like Maven Central. The development of frameworks like Goblin provides access to extensive dependency graph data. This enables real-time analysis of patterns and trends and broadens the scope of research. We can now explore the effects of dependency management practices, outdated libraries, and missed releases across various projects and domains globally.

Assessing how the number of dependencies affects the project is crucial due to its relevance to software maintenance and the ecosystem. Cataldo et al. [33] analyzed different types of dependency data from two independent software projects over an eight-year period. The analysis suggested software systems with a large number of dependencies, particularly logical and work dependencies, tend to be more complex and can lead to challenges in managing the development process. One of their research questions also revealed that as the inter-dependencies among tasks increase, the likelihood of defects in the software also rises. Tellnes [34] showed that the security and availability of a system are largely determined by the surrounding 'ecosystem' of dependencies. Prana et al. [35] highlights the importance of managing the number of dependencies and performing timely updates.

Several studies have demonstrated the necessity of specific metrics for quantifying dependency freshness and how it evolves. Cox et al. [5] performed correlation and longitudinal analysis to investigate the relationship between dependency freshness and known security vulnerabilities and to assess the variability of the metric over time, respectively. Kula et al. [36]

emphasized the importance of keeping dependencies updated, proposing a Software Universe Graph (SUG) to model dependency relationships and provide metrics for assessing update needs. Jafari et al. [37] investigated how different package characteristics can influence the predicted update strategy and found dependent count to be one of the highest influencing features. The results of Zerouali et al. [38] show the strong presence of technical lag and reluctance caused by the specific use of dependency constraints.

After evaluating the relevant studies, we find that there is room for further contributions in understanding the characteristics of dependencies and the freshness of projects. While existing literature explores dependency management, most studies focus on broad trends such as dependency growth, versioning policies, or security vulnerabilities. This leaves gaps in understanding whether there is a direct connection between the number of dependencies and the freshness of software projects, especially in the Maven ecosystem. Limited attention has been given to evaluating how up-to-date the dependencies in the latest releases are, which we aim to address in our study.

VI. Conclusion

This study presents an empirical analysis of dependency management in the Maven Central ecosystem, driven by two core research questions: whether projects with more dependencies are more likely to miss releases, and to what extent the dependencies of the latest releases are outdated. We use quantitative insights to analyze 100,000 libraries and over 1,000,000 dependencies. Our findings reveal that projects with fewer dependencies are more likely to miss releases, while projects with more than 200 dependencies tend to have fewer missed releases. We also find that the dependencies in the latest releases are generally up-to-date, indicating proactive management in current software development practices. With the increasing complexity of software ecosystems, this research provides actionable insights into the challenges of dependency management. Understanding patterns of missed releases and outdated dependencies can inform strategies to improve release reliability and dependency maintenance. However, it is important to acknowledge limitations such as dataset constraints and the scope of library selection, which may not fully capture the broader ecosystem of dependencies. Additionally, we did not include qualitative data, such as developer interviews or case studies, which could provide deeper insights into the challenges and strategies behind dependency management.

Future work could explore dependency management practices across diverse ecosystems or investigate the role of external factors, such as developer collaboration like direct surveys or interviews with them and release policies, in shaping dependency health. Real-time modeling of dependency release trends could also reveal evolving practices in managing software ecosystems. Ultimately, this research highlights key dependency trends and their implications, contributing to more resilient and efficient software engineering practices.

REFERENCES

[1] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *Journal of the Association for Information Systems*, vol. 11, no. 12, pp. 868–901, 2010.

[2] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–44, 2013.

[3] S. Raemaekers, A. Van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 221–224.

[4] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: a temporal graph-based representation of maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 344–348.

[5] J. Cox, E. Bouwers, M. Van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 109–118.

[6] D. Jaime, J. E. Haddad, and P. Poizat, "Goblin: A framework for enriching and querying the maven central dependency graph," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 37–41.

[7] D. Jaime, "Goblin: Neo4j maven central dependency graph," https://zenodo.org/records/13734581, Aug 2024.

[8] B. Chowdhury, M. Rabbi, S. Hasan, and M. Zibran, "Replication package," 2024. [Online]. Available: https://doi.org/10.6084/m9.figshare.27984506.v2

[9] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, pp. 381 – 416, 2017.

[10] Y.-C. Chen, "A tutorial on kernel density estimation and recent advances," *Biostatistics & Epidemiology*, vol. 1, no. 1, pp. 161–187, 2017.

[11] J. Leskovec and C. Faloutsos, "Sampling from large graphs," pp. 631–636, 2006.

[12] P. Bhattacharya and I. Neamtiu, "Dynamic updates for web and cloud applications," pp. 21–25, 2010.

[13] W. B. Richmond, P. Nelson, and S. Misra, "An empirical analysis of software life spans to determine the planning horizon for new software," *Information Technology and Management*, vol. 7, pp. 131–149, 2006.

[14] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, "Lifetime extension of software execution subject to aging," *IEEE Transactions on Reliability*, vol. 66, pp. 123–134, 2017.

[15] M. Islam and M. Zibran, "How bugs are fixed: Exposing bug-fix patterns with edits and nesting levels," in *SAC*, 2020, pp. 1523–1531.

[16] M. Islam and M. Zibran, "What changes in where? an empirical study of bug-fixing change patterns," *ACM Applied Computing Review*, vol. 20, no. 4, pp. 18–34, 2021.

[17] A. Rajbhandari, M. Zibran, and F. Eishita, "Security versus performance bugs: How bugs are handled in the chromium project," in *SERA*, 2022, pp. 70–76.

[18] M. Islam and M. Zibran, "A comparative study on vulnerabilities in categories of clones and non-cloned code," in *IWSC*, 2016, pp. 8–14.

[19] M. Islam, M. Zibran, and A. Nagpal, "Security vulnerabilities in categories of clones and non-cloned code: An empirical study," in *ESEM*, 2017, pp. 20–29.

[20] M. Islam and M. Zibran, "On the characteristics of buggy code clones: A code quality perspective," in *IWSC*, 2018, pp. 23 – 29.

[21] M. Zibran and C. Roy, "Conflict-aware optimal scheduling of code clone refactoring," *IET Software*, vol. 7, no. 3, pp. 167–186, 2013.

[22] M. Zibran, R. Saha, C. Roy, and K. Schneider, "Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study," in *28th ACM Symposium on Applied Computing*, 2013, pp. 1123–1130.

[23] D. Alwad, M. Panta, and M. Zibran, "An empirical study of the relationships between code readability and software complexity," in *SEDE*, 2018, pp. 122–127.

[24] A. Champa, M. Rabbi, F. Eishita, and M. Zibran, "Are we aware? an empirical study on the privacy and security awareness of smartphone sensors," in *SERA*, 2023, pp. 287–294.

[25] A. Champa, M. Rabbi, M. Zibran, and M. Islam, "Insights into female contributions in open-source projects," in *20th IEEE International Conference on Mining Software Repositories*, 2023, pp. 357–361.

[26] M. Islam and M. Zibran, "Exploration and exploitation of developers' sentimental variations in software engineering," *Internation Journal of Software Innovation*, vol. 4, no. 4, pp. 35–55, 2016.

[27] M. Islam and M. Zibran, "Towards understanding and exploiting developers' emotional variations in software engineering," in *SERA*, 2016, pp. 185–192.

[28] M. Islam and M. Zibran, "Sentiment analysis of software bug related commit messages," in *SEDE*, 2018, pp. 3–8.

[29] M. Islam and M. Zibran, "A comparison of dictionary building methods for sentiment analysis in software engineering text," in *ESEM*, 2017, pp. 478–479.

[30] M. Islam and M. Zibran, "A comparison of software engineering domain specific sentiment analysis tools," in *SERA*, 2018, pp. 487–491.

[31] R. Joseph, M. Zibran, and F. Eishita., "Choosing the weapon: A comparative study of security analyzers for android applications," in *SERA*, 2021, pp. 51–57.

[32] D. Murphy, M. Zibran, and F. Eishita, "Plugins to detect vulnerable plugins: An empirical assessment of the security scanner plugins for wordpress," in *SERA*, 2021, pp. 39–44.

[33] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, 2009.

[34] J. Tellnes, "Dependencies: No software is an island," Master's thesis, The University of Bergen, 2013.

[35] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, pp. 1–34, 2021.

[36] R. G. Kula, C. D. Roover, D. M. German, T. Ishio, and K. Inoue, "Modeling library dependencies and updates in large software repository universes," 2017.

[37] A. Javan Jafari, D. E. Costa, E. Shihab, and R. Abdalkareem, "Dependency update strategies and package characteristics," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–29, 2023.

[38] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *International Conference on Software Reuse*. Springer, 2018, pp. 95–110.