Graph Based Augmentation for Dependency Management in NPM

Shaquille Pearson
s23pears@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

The rapid growth of software ecosystems and the increasing reliance on third-party dependencies have underscored the need for robust dependency management practices. Effective dependency management is critical to ensuring software stability, security, and maintainability. However, managing complex dependency relationships, resolving version conflicts, and addressing transitive dependencies remain significant challenges, particularly in ecosystems like Node Package Manager (NPM). Existing tools often fail to provide adequate visibility into dependency structures or detect issues such as unused dependencies and cyclic relationships. This study explores the potential of graph-based approaches to enhance dependency management by enabling deeper insights into dependency networks and identifying limitations in existing methodologies.

KEYWORDS

Dependencies, NPM, Graphs, Visualization, Conflicts, Versions, Transitivity, Ecosystems, Packages, Software

ACM Reference Format:
Shaquille Pearson. 2024. Graph Based Augmentation for Dependency Management in NPM. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Dependencies are the backbone of modern software development, enabling developers to integrate pre-built functionalities into their applications. Tools like NPM and Yarn simplify dependency management, allowing developers to specify requirements through files like `package.json` and retrieve packages from extensive repositories. NPM, the largest ecosystem with over four million packages, supports the rapid evolution of JavaScript projects by fostering code reuse and modular development [9, 11]. However, managing dependencies remains a complex task, as projects must navigate a growing web of interconnected components. Issues such as transitive dependency conflicts, version duplication, and unused dependencies persist as recurring challenges that hinder project stability and increase maintenance overhead [5, 12].

These challenges are exacerbated by the dynamic nature of dependency networks. Frequent updates and inconsistent adherence to versioning conventions like Semantic Versioning (SemVer) often introduce technical lag and complicate dependency resolution processes [1, 3]. Studies in related ecosystems, such as Java and Python, have shown that dependency management failures can significantly affect software quality. For instance, Decan et al. [5] and Zerouali et al. [12] identified the risks posed by vulnerabilities and outdated dependencies, while Cogo et al. [3] highlighted the impact of reactive downgrades. Despite these findings, the NPM ecosystem remains underexplored, leaving critical questions unanswered about its dependency landscape.

This study addresses the gaps in existing literature by analyzing dependency-related issues in NPM projects and evaluating the feasibility of graph-based approaches to enhance dependency management. Specifically, it seeks to answer the following research questions:

- **RQ1: How prevalent are dependency-related issues in NPM projects?**
  Results: The analysis revealed key limitations in current tools, such as their inability to detect transitive dependency, cyclic relationships effectively and the inability to visualize dependencies effectively.
- **RQ2: Is there a need for graph-based dependency management tools in NPM projects?**
  Results: Dependency analysis revealed that issues like version duplication and transitive dependencies were common, impacting 30% of projects. However, cyclic dependencies were rare, with less than 2% occurrence across datasets.

2 STUDY DESIGN

This study aims to evaluate the feasibility and effectiveness of graph-based methods in enhancing dependency management within the NPM ecosystem. By analyzing patterns and relationships among dependencies, we seek to identify prevalent gaps in current approaches and provide strategies to mitigate dependency-related challenges. NPM, as the largest and fastest-growing software package ecosystem [9], serves as the focal point of the research. It hosts over four million packages and offers a robust registry where packages are published and maintained [5].

This section outlines the project selection and data filtration processes (Section 2.1). We then provide an overview of the graph construction and querying methodology (Section 2.2 and 2.3).

2.1 Project Selection

This study focuses on dependency management challenges in NPM packages, selecting JavaScript projects that use NPM as a primary dependency manager. These projects explicitly define dependencies in the `package-lock.json` file, which specifies the required packages and their version constraints. **Figure 1** outlines the projects collection process.

Conference'17, July 2017, Washington, DC, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

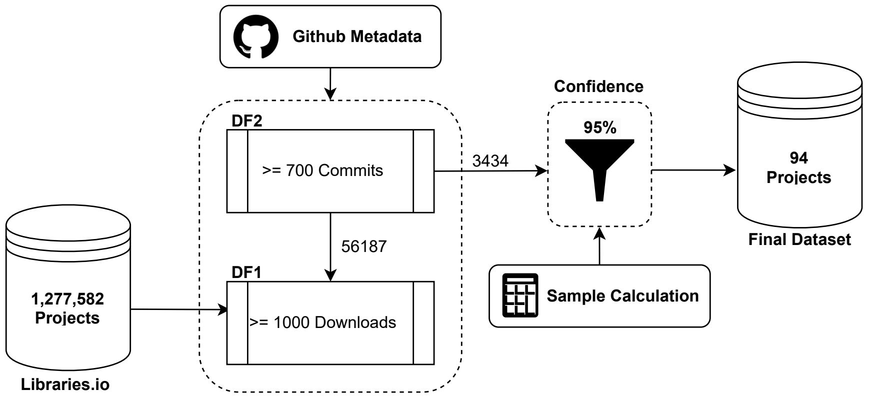

Figure 1: Overview of Project Selection

To curate a representative dataset, the process begins with Libraries.io [1], an open-source platform that aggregates a comprehensive repository of software projects across multiple package managers. Filtering for projects using NPM yields an initial dataset of 1.2 million projects. Recognizing that many of these projects lack maturity or the necessary complexity for meaningful analysis, a filtration step is implemented to refine the dataset. This step ensures the selection of projects that are both active and representative of the broader ecosystem. The filtration criteria are detailed in the following section.

- **Select significant projects (DF1).** Download count is an important metric reflecting a repository's relevance within the developer community, as it indicates how frequently a package is utilized by developers [7].
- **Filter mature projects (DF2).** A substantial commit history often signifies that a project is mature, stable, and dependable [7]. These projects typically have an active community of contributors and users, making them reliable options in the development ecosystem.
- **Sample projects.** To ensure the dataset is representative while maintaining feasibility, a sample size of 94 projects was determined based on statistical methods. The calculation, illustrated in **Figure 2**, utilizes a 95% confidence level, a margin of error of 10%, and a population size of 3,434 projects. This approach ensures a statistically valid representation [4]. The resulting dataset comprises 94 projects: 72 using package-lock version 2, 20 using version 1, and 2 with empty configurations.

Parameters:

Confidence Level (Z): 1.96 (95%)

Confidence Level Squared (Z^2): 3.8416

Proportion (p): 0.5

Margin of Error (e): 0.1

Population Size (N): 3434

Calculation:

Numerator: $N \cdot Z^2 \cdot p \cdot (1 - p)$

$$3434 \cdot 3.8416 \cdot 0.25 = 3303.93$$

Denominator: $(N - 1) \cdot e^2 + Z^2 \cdot p \cdot (1 - p)$

$$(3434 - 1) \cdot 0.01 + 3.8416 \cdot 0.25 = 35.3$$

Sample Size (n): $n = \frac{\text{Numerator}}{\text{Denominator}}$

$$n = \frac{3303.93}{35.3} \approx 93.6$$

Rounded Sample Size: $n = 94$

Figure 2: Sample size calculation (95% confidence level).

2.2 Graph Construction

Constructing the dependency graph involved representing packages as nodes and their relationships, such as dependencies, peer dependencies, and optional dependencies, as directed edges. This structure enables the analysis of intricate relationships among packages in the NPM ecosystem.

- **Nodes and Properties:** Nodes represent each package as a node with the following properties
  - **name:** Name of the package.
  - **version:** Version of the package.
  - **path:** File path of the package.
- **Relationships and Types:** Directed edges represent dependency relationships between packages:
  - **DEPENDENCIES:** Essential runtime requirements.

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

  - **PEERDEPENDENCIES:** Dependencies expected to be provided by the consumer.
  - **OPTIONALDEPENDENCIES:** Non-critical dependencies.
- **Justifications:**
  - **Directed edges:** reflect the dependency flow between packages, aiding in detecting transitive and cyclic dependencies [2, 5].
  - **Dependency types:** provides granularity for more nuanced analyses [6].

2.3 Query Justification

These queries leverage graph database capabilities to explore the intricate relationships and properties inherent in dependency networks. By addressing critical challenges such as detecting unused dependencies, identifying cyclic relationships, evaluating transitive dependencies, and calculating graph density, the queries aim to provide a comprehensive understanding of the dependency ecosystem. Each query was carefully crafted to align with the research objectives, ensuring that the extracted data highlights areas of potential improvement and supports the feasibility of graph-based solutions. The following queries not only target specific dependency management challenges but also provide metrics that underpin the findings of this study.

Transitive dependencies, packages that are not directly required by a given project but are introduced indirectly through the dependency graph, often complicate dependency resolution [2]. This query identifies all packages d that lie two or more hops away from a given package p, ensuring distinct nodes are counted to avoid overestimations. Such insights highlight the depth and complexity of dependency networks.

Cyclic dependencies occur when a package directly or transitively depends on itself, creating potential infinite loops during resolution and increasing complexity [5]. The query detects these cycles by finding paths where a package p leads back to itself, offering critical insights into problematic cycles.

Graph Density, which measures the proportion of actual connections to possible connections, captures the interconnectedness of the dependency network [6]. A dense graph may indicate tightly coupled ecosystems, while a sparse graph could suggest loosely connected packages.

Most Depended On Package Query, packages with the highest number of dependents, are critical for ecosystem stability [1]. Identifying these packages can guide priority maintenance and security auditing efforts.

Version Duplication, where different projects depend on varying versions of the same package, complicate dependency resolution [11]. By identifying such inconsistencies, this query sheds light on areas where alignment in versioning could improve maintainability and compatibility across projects.

3 STUDY RESULTS

(RQ1) Identify Gap in NPM Dependency Management Tools

A key focus of this study was to evaluate the limitations of current tools used for dependency management and to highlight how graph-based methodologies can address these shortcomings. **Table 1** provides a detailed summary of the identified gaps and potential graph-based solutions, with references where applicable.

A key focus of this study was to evaluate the limitations of current tools used for dependency management and to highlight potential improvements.

Key Findings:

- **Cyclic Dependencies:** Existing tools rely on manual checks for circular dependencies [2].
- **Unused Dependencies:** Unused packages are not fully flagged, causing inefficiencies [5].
- **Transitive Dependencies:** Tools lack clarity on nested dependencies [6].
- **Impact Analysis:** Updates lack comprehensive insight into cascading effects [1].
- **Version Resolutions:** Tools do not provide clear conflict visibility [5].
- **Visualization:** Visualization is limited, offering minimal analytical value [1].

(RQ2) Are graph-based tools needed for NPM dependencies?

Graph Density Comparison Graph density measures the extent of interconnections within a dependency graph, providing insight into the complexity and cohesiveness of the dependency network. **Figure 3** illustrates the comparison of graph density across projects for both Version 1 and Version 2 of the datasets.

The results indicate a noticeable difference between the two versions in terms of graph density. In Version 1, the graph densities exhibit lower values overall, with peaks ranging from 1,000 to 3,000, reflecting a more loosely coupled network. In contrast, Version 2 shows significantly higher densities for many projects, with several values exceeding 4,000, suggesting a tighter dependency network. This disparity may be attributed to additional relationships or updated dependency structures in Version 2.

Interestingly, certain projects in Version 2 display dramatic increases in graph density compared to their counterparts in Version 1. This highlights the evolution of dependencies over time, where newer or updated projects may integrate more interconnected components. Such growth in graph density could indicate increased complexity, posing challenges for dependency resolution and management.

Dependency Relationships The radar chart in **Figure 5** illustrates the differences between Version 1 (v1) and Version 2 (v2) of the analyzed dependency data across five key metrics: **Peer Dependencies**, **Cyclic Dependencies**, **Optional Dependencies**, **Unused Dependencies**, and **Transitive Dependencies**. The values are plotted as proportions relative to the total number of packages in each version, with averages displayed for additional context.

Comparison of Key Metrics:

- **Peer Dependencies:** In v1, the total number of peer dependencies recorded was **64**, while v2 shows no recorded peer dependencies. This discrepancy suggests a fundamental difference in how peer dependencies were captured or present between the two versions.
- **Cyclic Dependencies:** Both versions demonstrate a negligible number of cyclic dependencies, with v1 showing **6**

Table 1: Identified Gaps in Existing Dependency Management Tools

Gap Category	Description	Tools & Commands	Gap Identified
Cyclic Dependencies	Difficulty detecting circular dependencies[2]	npm, Yarn, npm-graph; No direct command	Manual detection required
Unused Dependencies	Ineffective identification of unused packages[3]	npm, npm-graph, Dependabot; `npm prune` removes unused from `node_modules` only	Untracked in `package.json`
Transitive Dependencies	Poor visibility into nested dependencies[4]	npm, Yarn, Dependabot; No native command	Hidden at multiple nested levels
Impact Analysis	Hard to see how updates affect downstream projects[5]	npm, Dependabot; `npm outdated` lists only outdated packages	No insight into cascading effects
Version Resolutions	Limited enforcement of version ranges[6]	npm, Yarn; `yarn resolutions` pin versions	No direct conflict visibility
Visualization	Lack of intuitive dependency visualization[7]	npm, npm-graph, Yarn; `npm-graph` shows trees	Limited analytical insights

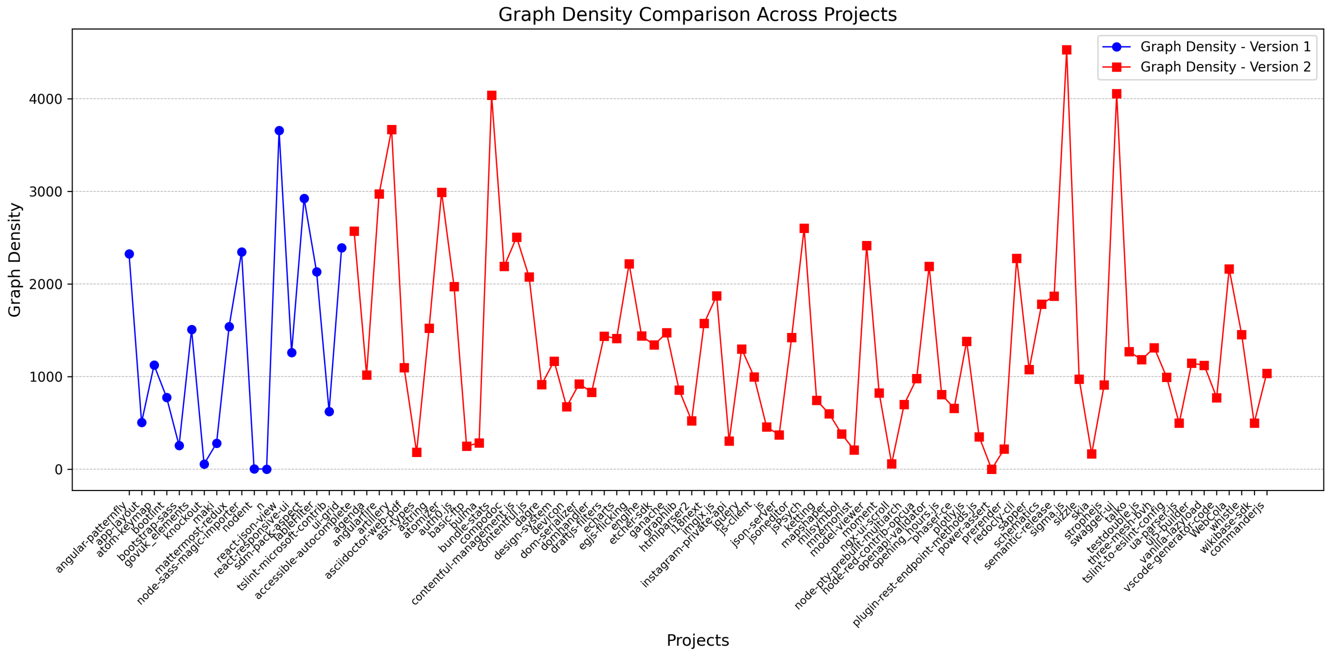

Figure 3: Graph Density Across Projects

and v2 showing **0**. This low occurrence suggests that cyclic dependencies are uncommon across the analyzed projects.

- **Optional Dependencies:** The v1 dataset recorded **19** optional dependencies, compared to none in v2. This highlights a potential limitation in v2's dependency recording or a genuine difference in the package ecosystem between the two versions.
- **Unused Dependencies:** The most striking difference lies in unused dependencies. In v1, **346** unused dependencies were recorded, compared to **252** in v2. This represents a notable reduction of **27%**, indicating a cleaner dependency graph in v2 or an improvement in dependency management practices.
- **Transitive Dependencies:** The v1 data recorded **148** transitive dependencies, whereas v2 recorded **158**, marking a slight increase. This could reflect deeper or broader dependency graphs in the v2 projects.

Version Duplication vs. Most Depended-On Packages The graphs illustrate the relationship between version duplication and the most depended-on packages across projects, with results segmented by **Version 4** and **Version 6** of `package-lock.json`. This analysis is essential for understanding how version inconsistencies propagate within the dependency network and their potential implications for widely used packages.

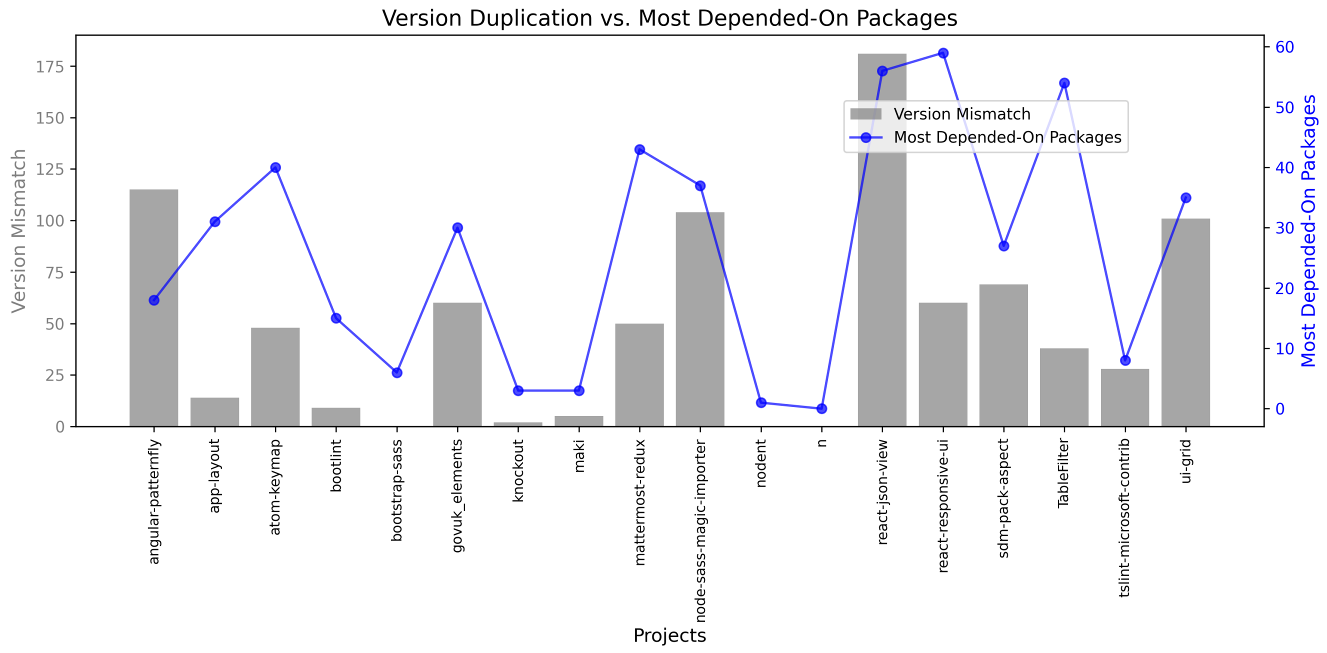

Figure 4: Package-lock Version 1

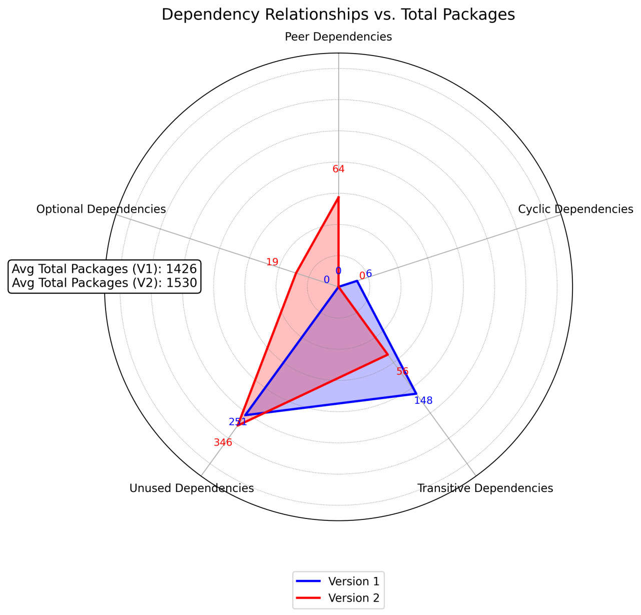

Figure 5: Types of Dependencies in Package-lock Versions

Comparative Insights.

- **Version Duplication:** Version 1 results exhibit more significant peaks, suggesting projects in this dataset face greater challenges with dependency conflicts. In contrast, Version 2 data shows fewer and less severe duplication, indicating possible improvements in dependency resolution practices.
- **Most Depended-On Packages:** The centrality of certain packages remains evident in both versions, emphasizing their critical role in the ecosystem. However, Version 2 exhibits a slightly more balanced dependence distribution, potentially reflecting improved modularity or reduced reliance on single packages.
- **Alignment:** In both graphs, high-dependence packages correlate with notable version duplication, reaffirming the need for careful dependency management in projects with widespread use.

4 PRACTICAL IMPLICATIONS

This section presents key practical insights derived from the study, offering actionable recommendations for developers and researchers.

Developers:

- **Observation 1: Graph-Based Dependency Visualization.** The use of graph-based methodologies provides developers with a clearer and more actionable understanding of complex dependency networks. Developers should adopt tools that offer graph-based visualizations to identify unused, transitive, and cyclic dependencies, enabling more efficient project maintenance and dependency pruning.
- **Observation 2: Addressing Version Duplication.** Version duplication across projects continue to present significant challenges. Developers should incorporate automated version conflict resolution systems, leveraging tools that detect

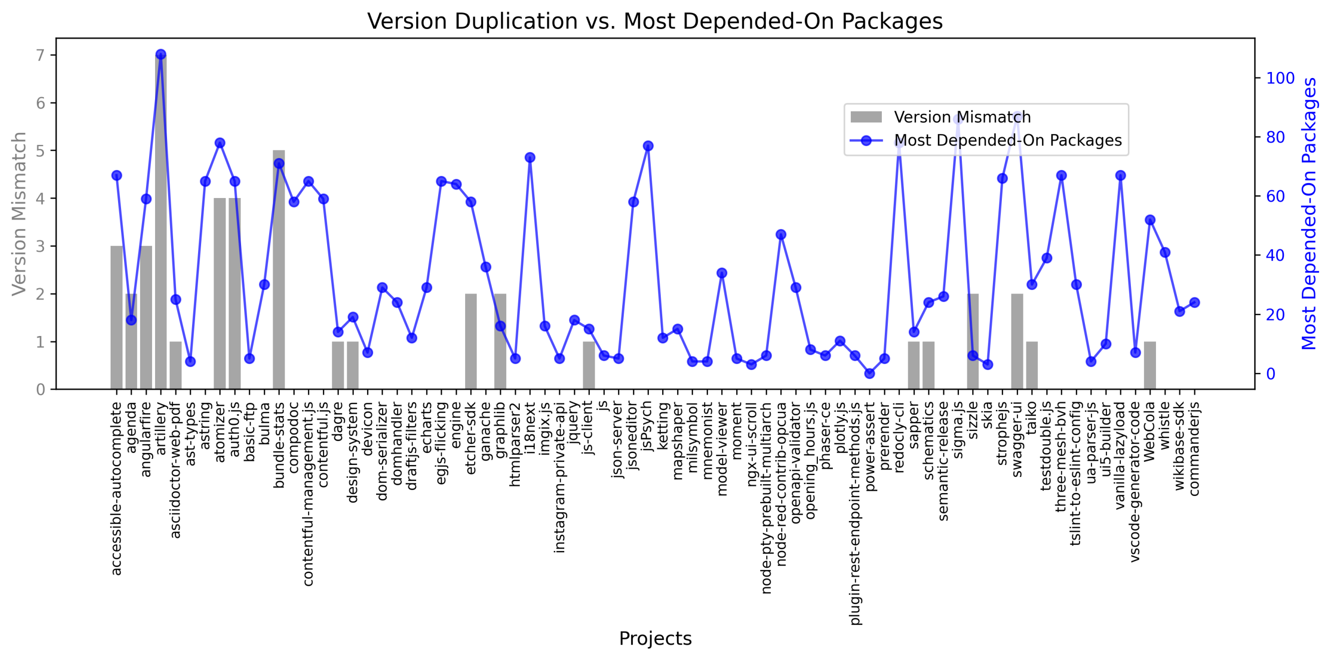

Figure 6: Package-lock Version 2

and recommend compatible dependency versions. Proactively managing dependency updates can reduce the risk of breaking changes and improve project stability.

- **Observation 3: Peer Dependency Management.** Peer dependencies, often neglected or misconfigured, lead to critical compatibility issues. Developers should maintain well-documented peer dependency requirements and test the integration of these dependencies rigorously during build and CI/CD processes. Tools that flag potential peer dependency issues early in the development lifecycle are recommended.

Researchers:

- **Observation 4: Advancing Graph Algorithms for Dependency Analysis.** The study highlights the utility of graph-based techniques in analyzing complex dependency structures. Researchers should explore novel graph algorithms tailored to specific dependency-related issues, such as detecting critical nodes, optimizing graph density, or identifying redundant dependencies.
- **Observation 5: Understanding Ecosystem Trends.** The analysis of dependency ecosystems across multiple versions reveals patterns in dependency evolution and management practices. Researchers should leverage these patterns to study the scalability of dependency management strategies, focusing on identifying best practices for handling rapidly growing ecosystems like npm.
- **Observation 6: Building Automated Tools.** The findings from this study can inform the development of automated tools that proactively monitor and resolve dependency-related failures. Researchers should focus on integrating early detection systems and predictive analysis models that help developers anticipate and address dependency conflicts before they manifest as build failures.

5 FUTURE WORK

A functional-level dependency graph offers the potential to address several critical challenges. By capturing the internal dynamics of packages, this approach could provide a more detailed view of unused or redundant functions, facilitating the identification of inefficiencies and opportunities for optimization [2, 5]. Furthermore, functional-level analysis could enhance conflict resolution by identifying specific functions or APIs responsible for incompatibilities, enabling developers to implement targeted fixes [1, 11]. Additionally, modeling dynamic interactions between functions could inform real-time adjustments to dependencies, fostering greater modularity and flexibility in software projects [3, 10].

Constructing such graphs would require advanced techniques, such as Abstract Syntax Tree (AST) analysis or bytecode inspection, to accurately extract and model inter-function relationships. Additionally, the increased granularity and scale of functional dependency graphs pose challenges in visualization and analysis, demanding innovative solutions to effectively represent and process complex data [8]. Incorporating machine learning techniques and graph query languages like Cypher could further enhance the capabilities of functional dependency tracking tools.

6 CONCLUSION

The dynamic and interconnected nature of the NPM ecosystem presents a unique set of challenges in dependency management, including version mismatches, cyclic dependencies, and redundant or unused dependencies. This study analyzed these issues through the lens of graph-based methods, offering a structured approach to model, query, and interpret complex dependency relationships. By constructing dependency graphs and leveraging targeted queries, this work provided a comprehensive understanding of the ecosystem's structural and behavioral characteristics.

The analysis of dependency relationships revealed critical gaps in existing tools, such as limited visibility into nested dependencies and insufficient detection of unused or cyclic dependencies. These gaps underscore the need for more advanced tools that leverage graph-based methods to provide granular insights and actionable solutions. The comparison of package-lock versions further highlighted significant differences in dependency structures, illustrating the evolution of dependency management practices over time and the potential for graph-based tools to adapt to these changes.

While this study focused on package-level relationships, future work can extend this approach to the functional level, enabling deeper insights into dependency behaviors and facilitating more precise conflict resolution. The integration of graph databases and advanced querying techniques, such as Cypher, demonstrates the feasibility and value of applying graph-based approaches to dependency management.

REFERENCES

[1] M. Alfadel, D. E. Costa, E. Shihab, and B. Adams. 2023. On the discoverability of npm vulnerabilities in Node.js projects. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023).

[2] C. Bogart, C. Kastner, J. Herbsleb, and F. Thung. 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.

[3] Filipe Roseiro Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. 2021. An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2457–2470. https://doi.org/10.1109/TSE.2019.2952130

[4] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[5] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.

[6] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24 (2019), 381–416.

[7] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[8] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1021–1031.

[9] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 351–361.

[10] E. Wittern, P. Suter, and S. Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the International Conference on Mining Software Repositories*. 351–361.

[11] A. Zaimi, A. Ampatzoglou, N. Triantafyllidou, A. Chatzigeorgiou, et al. 2015. An empirical study on the reuse of third-party libraries in open-source software development. In *Proceedings of the Balkan Conference on Informatics Conference*. 1–8.

[12] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*. Springer, 95–110.