Exploring Dependency Related Breakages in the NPM Ecosystem

Shaquille Pearson & Mahmoud Alfadel

Abstract—NPM stands as one of the world's most extensive software registries and is the home to an immense array of projects spanning both professional and amateur domains. Within this vast ecosystem lie millions of projects interlinked by dependencies, showcasing the dynamic nature of modern software development. The sheer magnitude of interconnected projects underscores the critical significance of effective dependency management. As developers leverage and contribute to this expansive repository, ensuring the reliability of software becomes a paramount challenge. Effective management of dependencies becomes a crucial element in preserving the integrity, functionality, and overall reliability of software projects embedded within NPM's ecosystem.
Our research delves into dependency-related build failures within the NPM ecosystem by analyzing **24** JavaScript projects. Rigorous project selection criteria, encompassing factors like popularity, activity, autonomy, and dependency installation steps, ensured a representative dataset. Employing Git and act, we traced commits affecting package.json files and locally reproduced builds for detailed analysis. Through manual classification of build outcomes, specifically targeting dependency-related failures, our study addresses three key research questions. Firstly, we emphasize the pivotal role of essential build steps, identifying that approximately **11.7%** of project failures stem from dependency-related issues, highlighting the pressing need for improved dependency management practices. Secondly, we underscore the impact of automation in swiftly resolving simpler dependency-related issues, with the quickest fixes taking only **2 minutes**. Conversely, longer fix times, extending up to **5 months**, emphasize the intricate and time-intensive nature of manual interventions in software development. Lastly, our findings reveal that dependency-related failures, constituting **60%** of total issues, are systemic challenges affecting a diverse array of software projects. Among these challenges, peer dependency conflicts emerge as the most prevalent issue, underscoring their critical role in build failures.

Index Terms—NPM, Yarn, Build

1 Introduction

Modern software systems are built using existing third-party packages, modules, components, or libraries, collectively referred to as dependencies [20] [18]. These dependencies play a fundamental role in the development process, providing pre-built and reusable functionalities that contribute to the efficiency, scalability, and overall robustness of software applications. They are usually linked to projects via symbolic declarations in modern package managers such as npm or yarn for javascript [1] [2]. This allows them to be downloaded from remote repositories at build time. By using version constraints developers can control which versions are compatible with the parent project which can provide significant flexibility in the dependency management process [1].

Maintaining compatible package versions takes up a significant amount of time and developers don't always adhere to `NPM's` version management system "`SemVer`" [10] [14], which introduces technical lag [21]. The dynamic nature of dependencies introduces a continuous evolution process, further exacerbating the issue of technical lag. Consequently, the delay in updating to newer versions may lead to unexpected behaviors within software systems, and in more severe cases, it can even result in system breakages.

While prior research has extensively explored build failures linked to software dependencies, focusing on languages like Java, C++, and Python, our study uniquely hones in on the `NPM` ecosystem. Seo et al [16], Horton and Parnin [9], and Mukherjee et al [13] showcased the widespread impact of dependencies on builds. In the domain of dependency studies, Decan et al [6] and Zerouali et al [21] unveiled insights into vulnerability consequences and technical lag, respectively. Meanwhile, Cogo et al [4] emphasized reactive downgrades. However, our contribution surpasses existing work by providing a nuanced analysis specific to `NPM`, providing a taxonomy of breakages and breakage time severity while also grouping them according to respective `NPM` phases.

In this study, we present a comprehensive analysis of build failures related to `NPM` package management. Our focus on the NPM ecosystem allows us to offer targeted insights into dependency challenges, addressing the unique intricacies of dependency-related breakages. More specifically, we set out to address the following research questions:

(RQ1) How prevalent are dependency-related breakages?
Motivation: The outcomes of our investigation hold substantial value for developers, project managers, and teams involved in `NPM` projects. By providing a thorough comprehension of the challenges linked to dependencies, our study empowers stakeholders to implement proactive strategies. This, in turn, facilitates the improvement of project reliability, reduc-

• *S. Pearson and M. Alfadel are with the David R. Cheriton School of Computer Science, University of Waterloo, Canada.*
E-mail: s23pears@uwaterloo.ca,malfadel@uwaterloo.ca

Manuscript received date; revised date.

1. https://www.npmjs.com/
2. https://yarnpkg.com/

tion of build failures, and the cultivation of a more streamlined and resilient development process.

Results:

The analysis of 24 NPM projects revealed that the inclusion of essential build steps, notably installation and script/build, played a pivotal role in the assessment of dependency-related issues. A notable 11.7% of project failures were linked to challenges in managing dependencies. The results emphasize the need for enhanced dependency management practices within the NPM ecosystem to improve overall project reliability and minimize failure rates.

(RQ2) How long do these breakages persist?

Motivation: For developers, acknowledging the influence of automation emphasizes the importance of automating routine tasks in dependency resolution. This understanding facilitates process streamlining, reducing disruptions, and improving overall project efficiency. Researchers, on the other hand, extract valuable insights into prevalent issues, laying the foundation for crafting automated analysis tools. These tools, guided by recognized patterns, can offer early detection and resolution suggestions, making substantial contributions to project stability and effectiveness across its lifespan

Results:

We analyzed the time required to address all **27** unique dependency-related failures. This examination reveals an average fix time of **12 days**. Remarkably, the swiftest resolution occurred within **2 minutes**, highlighting the efficiency of automated responses. Conversely, the lengthiest fix time extended to **5 months**, emphasizing the intricate and time-consuming nature of manual interventions in software development. The significant variability in fix times can be attributed to the diverse approaches adopted, with automated systems demonstrating rapid responses, especially in cases involving straightforward version changes. To validate our findings, we employed cross-reference validation, assigning the same manual analysis task to another author, thus reinforcing the reliability and consistency of our results.

(RQ3) What are the breakage categories?

Motivation: By categorizing distinct types of dependency-related failures and identifying their prevalence, developers gain actionable insights to enhance dependency management strategies. Project managers can benefit from a systemic understanding of these issues, enabling them to allocate resources effectively and implement preventive measures. For researchers, these findings contribute to the broader understanding of challenges in NPM ecosystems, guiding the development of targeted tools and methodologies for improved software reliability and resilience.

Results:

The analysis identified seven distinct types of dependency-related failures impacting thirteen projects. Peer dependency conflicts, constituting **60%** of failures, underscore the need for effective resolution strategies. Lock file version mismatches and issues like incompatible node versions follow closely, posing risks during project builds. The exploration of failure stages in the NPM installation process highlights their critical impact, especially during the NPM install dependency resolution phase.

In summary, our study unravels crucial insights for developers, project managers, and researchers alike. The spotlight on essential build steps reveals an **11.7%** incidence of project failures linked to dependency-related issues, demanding immediate attention to fortify dependency management practices within the NPM ecosystem. Delving into fix times exposes the intricate dance between automation and manual intervention, with rapid **2-minute** resolutions contrasting against the complex **5-month** endeavors. Beyond these temporal dynamics, our analysis sheds light on the systemic nature of dependency-related challenges, particularly the dominance of peer dependency conflicts at **60%**. These findings not only offer practical strategies for immediate implementation but also fuel the broader discourse on fortifying software reliability and resilience in the ever-evolving NPM terrain.

Paper organization. The remainder of the paper is organized as follows. **Section ??** provides a background for the work. **Section ??** outlines our motivation. **Section 2** provides an overview of the design of our study, while Section 3 presents the results. **Section 4** discusses the broader implications of our observations. **Section 5** presents the threats to the validity of the study. **Section 6** situates this work with respect to the literature, while **Section 7** draws conclusions and proposes directions for future work.

2 Study Design

Our study aims to provide insights into the prevalence, duration, and patterns of dependency-related build breakages in software projects, To provide insights that enable more effective strategies for managing and mitigating these challenges. We focus on NPM packages (dependencies) that are available from the NPM ecosystem, which is the largest software package ecosystem in the development landscape [19]. To date, NPM hosts more than four million packages and has the highest rate of growth in terms of packages among all known programming languages.[3] NPM has a registry where packages are published and maintained [17].

In this section, we present our approach to project selection and data filtration (Section 2.1). Then, we provide an overview of our approach to running and analyzing the builds of the selected projects (Section 2.2 & Section 2.3).

2.1 Collecting Project Dataset

Since we focus on dependency-related build breakages in NPM packages, we opt to select JavaScript projects that adopt NPM to manage their dependencies. Such projects must specify their dependencies in a `package.json` file, which lists the packages upon which the project depends, as well as their versioning constraints.

3. https://libraries.io/NPM

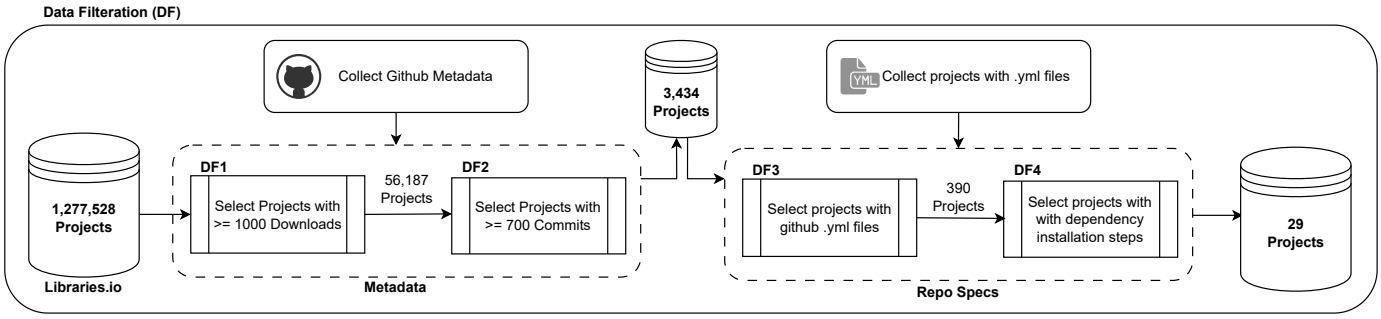

Fig. 1: Overview of the project selection process.

We begin our process with Libraries.io [12], an open-source web service that hosts an extensive repository of software projects spanning various package managers. Upon filtration of projects that adopt NPM, we find 1.2 million projects that adopt NPM. Since many of these projects are not yet mature or of sufficient complexity to warrant analysis, we apply several criteria for selecting a set of projects that are representative of our study analysis. **Figure 1** provides an overview our project selection process. We describe each data filtration **(DF)** step below:

- **Select popular projects (DF1).** The download count serves as a key indicator of a repository's value within the developer community because it reflects the actual usage of a package by developers [11]. We select a threshold of 1,000 downloads builds because it is closer to a "knee" in the curve (see **Figure 2**). Selecting this threshold further reduces the number of projects in the dataset to **56,187** projects.
- **Filter large and active projects (DF2)** Projects with a substantial commit history are typically well-established, stable, and reliable [11]. They often boast a vibrant community of contributors and users, making them a trusted choice within the developer community. Hence, we choose 700 commits as a threshold for identifying mature projects. **Figure 3** shows the threshold plot with **3,434** projects remaining at the knee.
- **Ensure local build reproducibility (DF3)** We prioritize projects with Continuous Integration (CI) "build" jobs that operate autonomously, without relying on external dependencies or CI/CD credentials like GitHub secret tokens, see line *24* in **Listing 1**. By ensuring that the CI process does not need external credentials, we promote robust and reliable project builds and it helps to maintain the local reproducibility of builds using `act` [22]. `Act` allows us to run GitHub Actions workflows locally by simulating GitHub's execution environment, enabling us to run builds without committing or pushing changes.
- **Validating dependency usage (DF4)** We focus specifically on projects that incorporate dependency installation steps within their Continuous Integration (CI) workflows. This focus aligns with our primary objective, which is to study and analyze dependency-related build breakages in real-world software projects. Our approach involves examining each CI file individually to identify specific commands within the "build" job, such as `npm i`, `npm ci`, or `npm update`. See line *26* in **Listing 1**

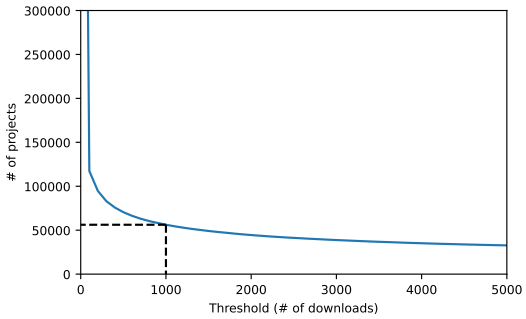

Fig. 2: Threshold analysis for the number of downloads.

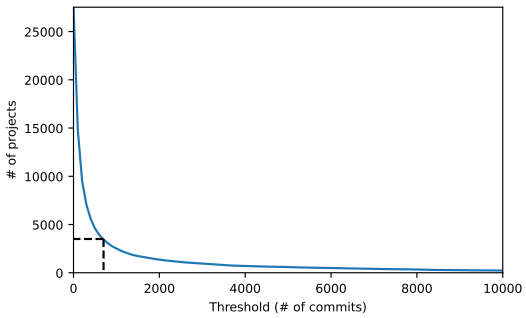

Fig. 3: Threshold analysis for number of commits

2.2 Build replay

After collecting the project dataset, we create local builds for each project. Our process of build replay comprises two main steps. Below, we explain each step.

Collect Dependency Commits. To investigate dependency-related breakages, we gather commits directly affecting the `package.json` file in our selected projects. Focusing on `package.json` commits is essential as these commits are likely to introduce problems associated with dependencies, since `Node.js` applications define their dependencies in `package. json` along with their version constraints [2]. We use `Git` commands to trace the version control system history of each project.

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      fail-fast: true

    steps:
      - uses: actions/checkout@v2
      - name: Cache node modules
        uses: actions/cache@v2
        env:
          cache-name: cache-node-modules
        with:
          path: ~/.npm
          key: ${{ runner.os }}-build-${{ env.
    cache-name }}-${{ hashFiles('**/package-
    lock.json') }}
          restore-keys: |
            ${{ runner.os }}-build-${{ env.
    cache-name }}-
            ${{ runner.os }}-build-
            ${{ runner.os }}-
      - uses: actions/setup-node@v2
        with:
          node-version: '16'
          cache: 'npm'
          github-token: ${{ secrets.
    GITHUB_TOKEN }}

      - name: Install dependencies
        run: npm i

      - name: Run Build
        run: npm run build:all
```

Listing 1: Example Build File

Build Reproduction. To replay builds, we use `act`, a tool for locally reproducing GitHub actions[4]. The tool interprets and executes actions defined in the project `.yml` files, generating build scripts and integrating with Docker to ensure emulation of GitHub action workflows locally [22]. Using the commits that impact the `package.json` file and `act`, we create isolated environments for each commit. `Act` reads GitHub Actions workflow files, generates tailored workflows for each commit, and executes them in separate Docker containers. Within these environments, we execute the designated "build" job using `act`, capture the output, and save it in individual text files. We then ensure a clean slate is prepared for the next build by resetting the environment after each commit with `git reset`, ensuring that our builds are not affected by noise.

2.3 Classifying Builds

In this section, we detail our approach to conducting an in-depth examination of the build logs produced for every project. Our objective is to delve into the prevalence and patterns of build failures attributed to dependencies. We analyze each log, aiming to gain insights into the reasons behind both failed and successful builds. By manually reviewing the logs, we aim to uncover trends that may not be apparent through automated processes alone.

4. https://github.com/nektos/act

(Manual Analysis). We consider the following to determine if a build is broken and the primary cause for the breakage:

- **Job.** This encompasses the execution of a series of predefined tasks or actions within a specified environment. These tasks may include building, testing, deploying, or performing other operations related to the project's development lifecycle.
- **Step.** This encompasses the execution of various commands essential for the build, test, and deployment processes.
- **Fail.** If any step within the job fails, we label the entire build as "failed."
- **Pass.** If all steps with the build job are successful then the build is labeled "passed".

3 Study Results

In this section, we provide a presentation of the outcomes derived from our study, specifically addressing the research questions at hand. We outline our approach, detailing the steps we undertook to investigate it. Subsequently, we share the observed results and findings that have emerged as a result of our research efforts. This approach allows for a clear and organized presentation of our findings, providing valuable insights into each research question's resolution.

(RQ1) How prevalent are dependency-related breakages?

RQ1: Approach. Our analysis involved the examination of a total of **24** individual projects. We followed the manual analysis methodology detailed in the previous section, executing builds for each commit in chronological order. We continued this process until the corresponding .yml file was no longer available or until the job configurations were no longer aligned with our predefined specifications.

RQ1: Results.

Figure 4 outlines our project selection, commits that touched package.json, and the number of builds we were able to reproduce. We observed a disparity between the number of builds we successfully reproduced and the total count of commits can be attributed to our specific project selection criteria. We deliberately focused on mature projects, which naturally accumulate a substantial number of commits over time. However, Continuous Integration and Continuous Deployment (Ci/CD) configurations may not be universally implemented right from a project's inception. In practice, Ci/CD pipelines and configurations tend to be introduced at later stages in a project's development, especially as the project grows and increases in complexity to avoid breakages and allow faster deployment, etc. [15]. Consequently, some of the initial commits lacked .yml configurations, resulting in a lower count of builds that we were able to reproduce. Furthermore, if the build process configuration didn't include typical steps such as installation and script/build [8], steps we opted to skip it.

Table 1 shows the percentage of failing and passing builds for the entire dataset. We found that **11.7%** of failures are due to dependency-related issues and another **53.5%** failed for other reasons making for a total of **65.2%** failures.

Project	Commits	Reproduced Builds
akita	188	20
atomizer	162	45
bowser	179	32
browser-sync	580	1
casparcg-connection	161	10
color-names	68	16
cytoscape.js	252	6
d3	457	28
express-openapi-validator	369	2
history	275	3
hooks	197	65
javascript-sdk	244	11
jscpd	280	49
maker.js	190	1
msgpack-javascript	166	18
ngl	225	9
node-fs-extra	194	2
nuxt	501	301
OverlayScrollbars	112	29
puppet	614	290
react-countup	577	3
reg-suit	251	14
twig.js	113	7
use-gesture	271	20
Total	**6823**	**982**

Fig. 4: Selected Projects

This underlines the impact that poorly managed dependencies can have on a project. This concept holds true in two cases: first, when dealing with breaking updates to dependencies, and second, when persisting with dependencies known to have defects [3] [7]. In the context of breaking updates, dependencies often evolve, introducing new features, bug fixes, and sometimes breaking changes. Failing to monitor these updates can lead to compatibility issues, unexpected behavior, or even system failures. On the other hand, continuing to employ dependencies with known defects or vulnerabilities can jeopardize the stability and security of a project. Ignoring these issues can result in system vulnerabilities, performance degradation, or even data breaches.

The presence of essential build steps, such as installation and script/build, was a key factor in determining if a build was included in the analysis. ***11.7%*** *of project failures are attributed to dependency-related issues which highlight the need for better dependency management practices.*

(RQ2) How long do these breakages persist?

RQ2: Approach. We initiated our analysis with the dependency-related breakages which accounted for **11.7%** of total failures. We then implemented additional filtering to pinpoint a selection of **27** distinct dependency-related breakages that reoccurred over time to account for the total **115** dependency failures. Concentrating on these specific, repetitive failures, we aimed to gain valuable insights into the challenges associated with the project's reliance on these dependencies. Our aim was to analyze the breakage time for each failure, to calculate this, we took the timestamp of the initial breaking commit and the corresponding local build time **(using act)** [22]. Subsequently, we moved forward in time, taking into account the duration until the fixing commit's build was completed. This time span encapsulated the period during which a failure was detected and subsequently fixed. By calculating these time differences, we were able to quantify the breakage time for each failure.

Category	#	%
Passing Builds	*456*	*46.5*
Failing Builds	*526*	*53.5*
Dependency Related Build Failures	*115*	*11.7*

TABLE 1: Reproduced Builds

RQ2: Results.

Figure 5, displays an analysis of the time to resolve build breakages originating from all 27 unique dependency-related failures. This chart illustrates the duration from the introduction of the breaking commit to the point where the build is successfully fixed. The average fix time for these incidents is **12 days**, providing an overview of the typical timeframe for resolving these issues. Notably, the most rapid fix occurred in a mere **2 minutes**, showcasing a highly efficient response, while the longest took **5 months**. The substantial contrast between the shortest and longest fix times can be attributed to the distinct approaches used in addressing issues. In cases with the quickest fixes, automated systems or **"bots"** were employed. These systems implemented some breaking changes and swiftly rectified problems in the following commit. In contrast, longer fix times were typically tied to human intervention and involved more complex issues, such as compatibility problems, or codebase conflicts within the dependencies. Automated fixes, on the other hand, usually revolved around straightforward version changes in the project's package.json file.

(Cross-Reference Validation) - To validate the findings presented by the first author, we assign the same manual analysis task to another author. By having a second author independently perform the same manual analysis, we aim to confirm the consistency and accuracy of the findings, strengthening the validity of our analysis.

The wide range of fix times for dependency-related failures highlights the impact of automation in quickly addressing simpler issues, with the fastest fixes taking just ***2 minutes****. On the other hand, longer fix times, up to* ***5 months****, emphasize the complexities and time required for manual interventions in software development.*

(RQ3) What are the breakage categories?

RQ3: Approach. We conducted a manual analysis of the generated builds and compared the results of both authors.

ID	Reason	Description	%
R1	Frozen Lock File Error	`Package-lock.json` is frozen, meaning it's not being updated correctly to reflect changes	*5.22%*
R2	Lock File Version Mismatch	Versions of dependencies in `package.json` don't match those in `package-lock.json`	*13.04%*
R3	Peer Dependency Conflict	Modules in package.json require incompatible versions of third-party packages	*60.86%*
R4	Incomplete Installation Error	Errors that occur during the downloading/installation of dependencies	*2.61%*
R5	Missing/Undefined Dependencies	Dependencies specified in the `package.json` are not installed/defined	*7.83%*
R6	Incompatible Node Version	Node.js version specified in the `package.json` file does not match the version installed	*7.83%*
R7	Enoent Error	A specified path file or directory does not exist in the file system	*2.61%*
ID	**Install Phase**	**Description**	
R1-R3	Resolving Dependencies	NPM parses the `package.json` file and constructs a dependency tree of all dependencies	
R4-R5	Fetching Dependencies	NPM downloads and caches all dependencies on the local machine	
R6	Installing Dependencies	NPM extracts the downloaded packages and installs them in the project directory	
R7	Custom Install scripts	NPM runs any custom install scripts defined in `package.json`	

TABLE 2: Dependency Related Breakage Errors

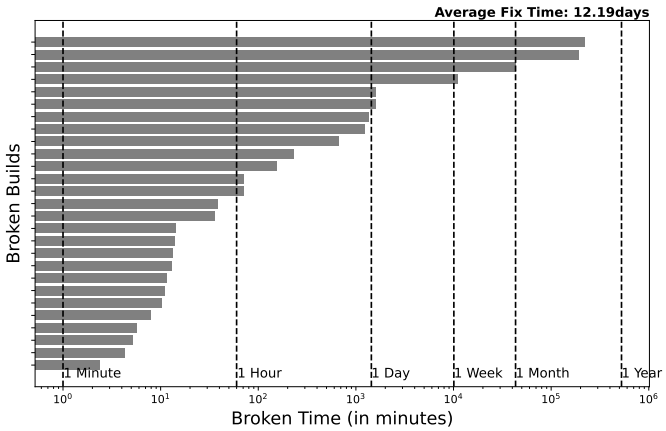

Fig. 5: Breakage Persistence

To quantify the agreement between the manual analysis and of both authors, we used the *Cohen Kappa coefficient* [5] to calculate the inter-rater agreement score which assesses how much agreement exists between two different raters or methods of categorization.

RQ3: Results. Table 2 presents a summary of the **7** distinct types of dependency-related failures that were identified during the analysis of builds. These failures had a significant impact on a total of thirteen different projects, indicating that they were not isolated incidents but rather systemic issues that affected multiple software projects. Notably, **peer dependency conflicts** emerged as the most prevalent issue, contributing a significant **60%** of the total dependency-related failures observed. The high level of peer dependency conflicts indicates their widespread occurrence and the critical need for effective resolution strategies. Following closely was **lock file version mismatches**, which represented the second most common type of failure. These mismatches can lead to discrepancies in expected and actual dependencies, causing unexpected issues during project builds. In third place were **incompatible node versions** and instances of **missing or undefined dependencies**. These inconsistencies can lead to runtime issues and disrupt the development workflow. On the other hand, the presence of missing or undefined dependencies can introduce complexities and errors in the build process of a project

Additionally, we identified the stages within the NPM installation process where the failures were encountered. It's important to note that the installation process consists of multiple phases, but we have exclusively highlighted those phases [5] [6] [7] that are relevant to our findings.

> *Dependency-related failures are not isolated incidents but systemic issues that impact a wide range of software projects. Peer dependency conflicts are the most prevalent, constituting* ***60%*** *of the total dependency-related failures, and most failures occur during the* ***dependency resolution*** *phase of* `npm install`.

4 Practical Implications

In this section, we outline a few *"lessons learnt"* that could have some practical significance for developers and researchers.

1. Developers:

- **Observation 1**, a wide range of fix times for dependency-related failures, from minutes to months. Automated systems, or "bots", demonstrated quick resolutions by implementing straightforward version changes. Developers should aim to automate repetitive tasks in the dependency resolution process, to minimize disruptions.
- **Observation 2**, peer dependency conflicts are the most problematic dependency-related failure. Developers should prioritize maintaining consistent versioning policies. Clearly defining and enforcing

5. https://docs.npmjs.com/cli/v6/commands/npm-install#synopsis
6. https://docs.npmjs.com/cli/v6/using-npm/scripts#npm-install
7. https://docs.npmjs.com/cli/v6/using-npm/scripts#life-cycle-scripts

version constraints for dependencies can mitigate the risk of conflicts arising from incompatible versions. Additionally regularly reviewing and updating dependencies to versions that are compatible with each other can prevent potential conflicts.

2. **Researchers:**

- **Observation 3**, the identified patterns of dependency-related failures in this study offer valuable insights into the challenges faced by software projects. These findings can be leveraged to guide the development of automated analysis tools that specifically target the most prevalent issues. Tools that provide early detection and resolution recommendations for these common problems can significantly contribute to the efficiency and stability of a project's lifespan.

5 Threats to Validity

In this section, we discuss various threats that could impact the validity of the research outcomes. By examining and addressing these potential limitations, we aim to enhance the credibility of our study. It is important to recognize that despite rigorous research methodologies, certain factors may introduce biases or constraints that could influence the applicability of the results. Identifying and openly discussing these threats provides transparency and helps readers to better understand the scope and limitations of the study.

5.1 Construct Validity

We focus on measuring dependency-related failures by examining commits that specifically touch the package.json file. However, it's essential to acknowledge that dependency issues can arise from a variety of changes beyond alterations to the package.json. Variations in configuration files, adjustments to the development environment, codebase refactoring, modifications to scripts and updates to plugins or extensions can all contribute to compatibility challenges.

However despite these broader potential contributors changes such as version updates, addition, or removal of dependencies directly influence the project's dependencies and, consequently, its build outcomes. This underscores the importance of honing in on `package.json` changes as the relevant metric for understanding the impact of dependency-related failures.

5.2 Internal Validity

A significant threat to the internal validity of our study stemmed from the inherent challenge of accessing CI/CD build data due to privacy and confidentiality constraints. The absence of direct access to these builds could manifest as potential inaccuracies in our analysis, as we might be unable to replicate the exact conditions of the build environments.

To tackle this threat, we employed the `act` tool, specifically designed for locally reproducing GitHub Actions. By utilizing `yml` files and Docker containers, we aimed to emulate the genuine build environments as closely as possible. This use of `act` served as a practical solution to the challenges associated with restricted access to CI/CD build data, ensuring that our locally reproduced builds maintained a high degree of fidelity to the original conditions and configurations.

5.3 External Validity

While our study centers on the Node Package Manager (NPM), it's crucial to recognize that narrowing the focus to a specific package manager might raise concerns about the generalizability of our findings to other ecosystems. However, this decision is mitigated by the fact that NPM stands as the largest software repository currently available, exerting a significant influence, particularly within the web development community.

The techniques and insights derived from our examination of NPM's practices and challenges are likely to have broader applicability across diverse software ecosystems. The web development community's extensive reliance on NPM, combined with its widespread adoption, positions it as a representative and influential case study. Therefore, while our study is rooted in the context of NPM, the implications and strategies identified are transferrable to other package management scenarios, enhancing the external validity of our findings.

6 Related Work

In this section, we explore the body of literature surrounding dependencies, their management, and the challenges they pose in real-world software projects. By examining the contributions of other scholars, we aim to contextualize our study within the broader landscape of software engineering. From dependency resolutions to the consequences of evolving software ecosystems, our exploration of related literature provides a foundation for our research. This involves recognizing existing knowledge but also pinpointing specific areas where our study contributes valuable perspectives for understanding and mitigating dependency-related breakages.

Build Failures In their investigation of Java and C++ build failures, Seo et al [16] uncovered that close to 50% of all build errors are attributed to software dependencies. Similarly, when assessing Python gists on GitHub, Horton, and Parnin [9] found that 52.4% of the gists failed to execute, primarily due to dependency errors. Mukherjee et al [13] analyzed 2,106 `BugSwarm` builds and found that 67.2% had dependency errors. As software dependencies significantly impact development workflows, the NPM ecosystem presents it's own unique challenges. Our study aims to provide targeted insights specific to NPM package management, optimizing installations, streamlining development, and enhancing overall reliability.

Dependency Studies Decan et al [6] explored the consequences of security vulnerabilities and observed that a majority of these vulnerabilities were of medium or high severity. The study further revealed that the fixes for vulnerabilities typically took a long time regardless of their severity but low-severity vulnerabilities took longer to be discovered. A notable finding was that over 50% of the

dependent packages were impacted by vulnerabilities originating from upstream sources. Zerouali et al [21] found that technical lag is introduced through the use of `package.json` constraints, specifically the caret symbol which prevents backward incompatible changes, which would explain the prevalence of medium to high-severity vulnerabilities since major updates aren't being applied promptly.

Cogo et al [4] findings resonate with some of the breakages we've encountered in our study, as they highlight that reactive downgrades are often triggered by package provider defects, unexpected feature changes, and incompatibilities. While their work provides valuable insights, our contribution seeks to augment this understanding by delving into more nuanced details of these breakages, providing a more granular categorization by grouping respective NPM installation phases to the breakages.

7 Conclusions

Previous investigations have emphasized the significant impact of software dependencies on build failures in various programming languages. In our study, we analyzed 24 JavaScript projects using NPM to delve into dependency-related build breakages. Our project selection criteria, including popularity, activity, autonomy, and dependency installation steps, ensured a representative dataset. Using `Git` and `act`, we traced commits affecting `package.json` files and locally reproduced builds for analysis. We then manually classified builds as "failed" or "passed" based on the build outcome, specifically focusing on the dependency-related failures.

- **(RQ1)** Essential build steps, especially installation, and script/build, play a crucial role in the identification of dependency-related issues. Approximately 11.7% of project failures are linked to dependency-related issues, underscoring the urgent need for enhanced dependency management practices.
- **(RQ2)** Automation significantly contributes to the swift resolution of simpler dependency-related issues, with the fastest fixes taking only 2 minutes. Longer fix times, stretching up to 5 months, highlight the intricate and time-intensive nature of manual interventions in software development.
- **(RQ3)** Dependency-related failures, constituting 60% of total issues, emerge as systemic challenges impacting a diverse range of software projects. Peer dependency conflicts are identified as the most prevalent issue, emphasizing their critical role in build failures.

Future Work. In future work, we aspire to leverage the insights gained from our study to develop a proactive tool aimed at enhancing dependency management practices. Specifically targeting essential build steps, such as installation and build, to provide developers with automated assistance in identifying and resolving dependency-related issues. Our goal is to streamline the software development process by reducing the occurrence of build failures and mitigating the impact of peer dependency conflicts

References

[1] M. Alfadel, D. E. Costa, M. Mokhallalati, E. Shihab, and B. Adams, "On the threat of npm vulnerable dependencies in node. js applications," *arXiv preprint arXiv:2009.09019*, 2020.

[2] M. Alfadel, D. E. Costa, E. Shihab, and B. Adams, "On the discoverability of npm vulnerabilities in node.js projects," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023. [Online]. Available: https://doi.org/10.1145/3571848

[3] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.

[4] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2457–2470, 2021.

[5] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[6] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191.

[7] 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, 2019.

[8] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 33–50, 2020.

[9] E. Horton and C. Parnin, "Gistable: Evaluating the executability of python code snippets on github," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 217–227.

[10] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3790–3807, 2021.

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

[12] J. Katz, "Libraries. io open source repository and dependency metadata (version 1.6. 0)," *URL: https://doi.org/10.5281/zenodo*, vol. 3626071, 2020.

[13] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 439–451.

[14] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "A large scale analysis of semantic versioning in npm," 2023.

[15] P. Rostami Mazrae, T. Mens, M. Golzadeh, and A. Decan, "On the usage, co-usage and migration of ci/cd tools: A qualitative analysis," *Empirical Software Engineering*, vol. 28, no. 2, p. 52, 2023.

[16] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case

study (at google)." New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2568225.2568255

[17] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1021–1031.

[18] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package managers," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 2020, pp. 228–237.

[19] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 351–361.

[20] A. Zaimi, A. Ampatzoglou, N. Triantafyllidou, A. Chatzigeorgiou, A. Mavridis, T. Chaikalis, I. Deligiannis, P. Sfetsos, and I. Stamelos, "An empirical study on the reuse of third-party libraries in open-source software development," in *Proceedings of the 7th Balkan Conference on Informatics Conference*, 2015, pp. 1–8.

[21] 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.

[22] H.-N. Zhu, K. Z. Guan, R. M. Furth, and C. Rubio-González, "Actionsremaker: Reproducing github actions," *ICSE-Companion. IEEE*, 2023.