# Analyzing Dependency Related Build Breakage in the NPM Eco-System

Shaquille Pearson
*Cheriton School of Computer Science*
*University of Waterloo*
Waterloo, Canada
s23pears@uwaterloo.ca

## I. Abstract

Dependency management is a critical aspect of software development as it helps ensure that a project builds and runs as expected. When a project depends on external libraries, it is important to keep track of these dependencies and their versions to avoid compatibility issues. If a dependency is updated and causes a build breakage, it can result in a number of problems. It can cause the build to fail, meaning development teams can't continue working until the issue is resolved. This can be time-consuming and can significantly delay the delivery of new features and updates. Furthermore, dependency related build breakages can also be difficult to diagnose and it can lead to unexpected behaviour in the software. [5] This can result in a loss of trust in the software and can cause a negative impact on the project's reputation. In this paper, we study dependency related build breakages in the NPM ecosystem. We investigate #245 GitHub projects that use NPM as their package manager and have at least #1000 downloads and #700 commits. Our results show that dependency related build breakages are a common occurrence in the NPM ecosystem with 14.85% of all builds failing. Furthermore, we found that the most common cause of dependency related build breakages is due to version conflicts among others such as source code changes, conflicting dependencies, missing dependencies and vulnerabilities. Our findings indicate that dependency related build breakages are a common occurrence in the NPM ecosystem and that they can have a significant impact on the stability and reliability of the software.

## II. Introduction

NPM (Node Package Manager) has become an essential part of modern web development, providing developers with a vast collection open-source packages that can be easily installed and integrated into existing and new projects. However, with the increasing complexity of these packages and their interdependencies, build breakages have becomes a common occurrence, leading to delays and errors in the software development process.

Build breakages can be caused by range of factors, such as conflicts between package versions, missing dependencies and outdated packages, among others. These issues can be difficult to detect and resolve, requiring significant time and effort from developers to troubleshoot and fix. Furthermore, the impact of build breakages can be far-reaching affecting not only the development process but also the quality and performance of the resulting software.

Given the critical role that NPM plays in modern web development, it is important to understand the root causes of dependency related build breakages and their impact in the NPM ecosystem. This paper aims to address this issue by analyzing a comprehensive set of NPM packages and builds using statistical techniques to identify the most common types of dependency related build breakages and their impact on the NPM ecosystem.

Through this analysis, we hope to provide a better understanding of the challenges faced by developers in managing dependencies and hopefully provide insights that can be used to improve the reliability and stability of NPM packages. By identifying the key factors that contribute to build breakages and offerings solutions to address these issues, we hope to help developers avoid these problems and improve the quality of the software they develop. In summary, this paper makes the following contributions:

- **Extent of Dependency Related Build Breakages (RQ1)** - Our investigation revealed that build breakages caused by dependencies are a pervasive issue in the NPM ecosystem, with 14.85% of all build in out dataset failing. This finding suggests that developers are frequently encountering problems relating to dependencies that prevent their code from compiling correctly or executing as intended. The high prevalence of build breakages highlights the importance of addressing this issue to improve the reliability and stability of software developed using NPM packages
- **Breakage Types (RQ2)** - Our findings suggest that version conflicts are the most common causes of dependency related build breakages. Other contributing factors include changes to source code, conflicting dependencies, missing dependencies and security vulnerabilities etc. Developers should be aware of these issues and take steps to carefully manage dependencies by monitoring updates and version changes and implement strategies to prevent conflicts and errors during the build process. These steps can improve the reliability and stability of software developed using NPM

## III. Background and Related Work

In recent years, the use of package managers like NPM has become ubiquitous in the software development industry. These package managers provide developers with a simple and efficient way to integrate pre-existing code into their projects, enabling them to build on top of a wide range of pre-built components without having to write everything from scratch. As a result, developers are able to create new software more quickly and efficiently than ever before. However, this ease of use comes with its own set of challenges. The number of packages available on NPM has grown exponentially in recent years, making it increasingly difficult for developers to keep track of all the dependencies their projects rely on. Furthermore, these packages are often interdependent, meaning that even small changes to one package can have ripple effects throughout the entire ecosystem. This complexity can make it difficult for developers to ensure that their projects remain stable and reliable.

Daniel Venturini et al. [6] analyzed a dataset of over 200,000 client packages and their dependencies to identify breaking changes that occurred over a period of 30 months. The study finds that breaking changes are a common occurrence, with nearly 30% of packages experiencing at least one breaking change during the study period. The paper also examines the impact of breaking changes on dependent packages, finding that these changes can result in significant build breakages and increased maintenance burden for developers. The authors conclude that developers need to be aware of the risks posed by breaking changes and implement strategies to mitigate their impact.

Ivan Pashchenko et al. [4] explores the challenges of dependency management in software development and its implications for security. The paper highlights the difficulty of managing dependencies, including the risk of introducing vulnerabilities through dependencies, the challenge of keeping dependencies up-to-date, and the lack of visibility into the security risks associated with dependencies. The study also identifies several strategies for improving dependency management, including better documentation and communication among developers, increased use of automated tools for dependency management, and improved training for developers on best practices for dependency management.

Christian Machon et al. [1] presents a novel approach to automatically repairing build breakages related to dependencies. The author highlights the challenges faced by developers in managing dependencies, such as version conflicts, and proposes an automated tool that can detect and resolve such issues. The tool analyzes the project's package.json file and generates a dependency graph to identify potential conflicts. It then suggests solutions, such as upgrading or downgrading dependencies, and automatically updates the project's package.json file to resolve the issue. The paper's findings suggest that such automated approaches can help improve the reliability and stability of software developed using NPM packages.

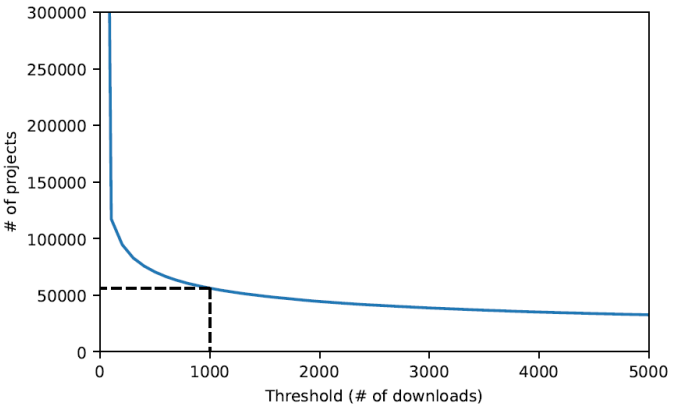


Fig. 1. Threshold Graph for Number of Downloads

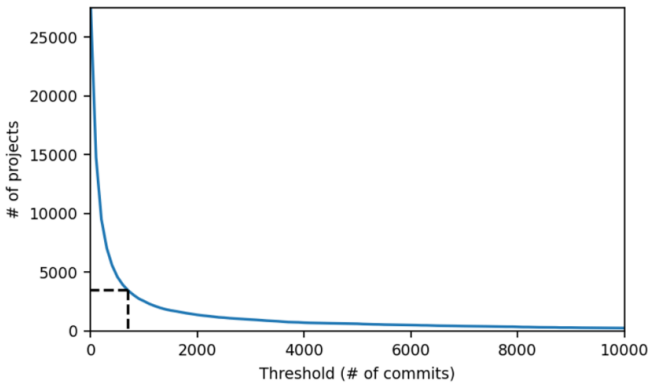


Fig. 2. Threshold Graph for Number of Commits

## IV. Experimental Design

The experimental design for our research involved selecting a dataset of all NPM packages present on GitHub for the year 2022 backwards. We then applied certain filtering criteria to narrow down our selection to only include projects that met certain minimum standards. Specifically, we chose to only select projects that had a minimum of 1000 downloads and 700 commits. After filtering our dataset, we then selected projects that used either CircleCI, Travis-Ci or GitHub Actions for their CI/CD pipeline which left us with 617 projects. This allowed us to focus our analysis on projects that were actively being developed and maintained, and that used popular tools for continuous integration and deployment.

To determine the threshold values for our filtering criteria, we generated threshold graphs at each filtration point. These graphs, displayed in **Figures 1 and 2**, helped us to determine the optimal values for the number of downloads and commits required for inclusion in our analysis. By selecting projects that met these minimum standards, we were able to ensure that our sample set was representative of actively developed and popular NPM packages on GitHub.

After selecting the relevant projects using the criteria mentioned in the previous section, the next step in our experimental

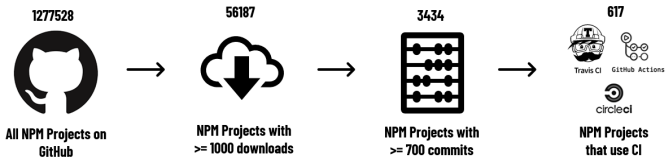


Fig. 3. An Overview of the selection Process


Fig. 4. An Overview of the Breakage Inducing Commit Collection Process

design was to collect all commits that involved changes to the package.json file. The package.json file is a critical file in NPM projects that contains information about the project's dependencies, scripts, and metadata. To determine the status of each commit, we utilized the GitHub API, which allows us to access data about GitHub repositories and their associated commits. Specifically, we checked the commit status using the API, which would return a value of either 'success' or 'failure'. A commit's status would only be returned if it was involved in one of the jobs for the build process, and a failure status indicates that the build failed.

When a build fails, the reason for the failure may not always be clear, and it could be due to a variety of reasons such as changes to the source code, issues with the build environment, or problems with the dependencies. To determine if the build failure was related to some dependency issue, we had to manually analyze the build log for each commit. This involved reviewing the logs either through the CI/CD tool's web interface or the build log file to identify any errors or warnings related to dependencies.

We looked for specific indicators such as error messages related to missing dependencies, version conflicts, or package installation issues. By analyzing the build log, we were able to determine if the build failure was related to a dependency issue or some other problem. If the build failure was due to


```
"state": "failure",
  "statuses": [
    {
      "url": "https://api.github.com/repos/suguru03/aigle/statuses/f48d8bcd0d4ac4d02c1606cfa0a1bf265c382b17",
      "avatar_url": "https://avatars.githubusercontent.com/oa/1508?v=4",
      "id": 10457852420,
      "node_id": "MDEzOlN0YXR1c0NvbnRleHQxMDQ1Nzg1MjQyMA==",
      "state": "error",
      "description": "The Travis CI build could not complete due to an error",
      "target_url": "https://travis-
          ci.org/github/suguru03/aigle/builds/717089397?utm_source=github_status&utm_medium=notification",
      "context": "continuous-integration/travis-ci/push",
      "created_at": "2020-08-11T23:10:06Z",
      "updated_at": "2020-08-11T23:10:06Z"
    },
  ],
```


Fig. 5. JSON Response from GitHub API with 'failure' status

```
Error: Cannot find module 'chalk'
Require stack:
- /home/circleci/project/scripts/jest/jest-cli.js
    at Function.Module._resolveFilename (node:internal/modules/cjs/loader:933:15)
    at Function.Module._load (node:internal/modules/cjs/loader:778:27)
    at Module.require (node:internal/modules/cjs/loader:1005:19)
    at require (node:internal/modules/cjs/helpers:102:18)
    at Object.<anonymous> (/home/circleci/project/scripts/jest/jest-cli.js:4:15)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12) {
  code: 'MODULE_NOT_FOUND',
  requireStack: [ '/home/circleci/project/scripts/jest/jest-cli.js' ]
}
error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
```

Fig. 6. Failing Build Log for a Dependency Related Breakage (Missing Module)

some other issue, such as a syntax error in the code or an issue with the build environment, we excluded it from our analysis of dependency-related build breakages. **Figure 5** shows an example of a build log that was due to a dependency issue.

## V. Experimental Results

**RQ1**: In the process of checking each commit, it was observed that some commits returned a pending status, which indicated that they were not involved in any builds. Therefore, such commits were dropped from the analysis. After this step, there were 245 projects with build-triggering commits that were included in the analysis. We found that dependency related build breakages were present in 14.8% of builds for a total of 9182. **Figure 7** shows the total successful/failed builds for each platform.

The fact that dependency-related build breakages were present in nearly 15% of builds is a significant finding, as it underscores the importance of managing dependencies effectively in software projects. These types of breakages can have a significant impact on the stability and functionality of a project, potentially leading to downtime, loss of data, or other serious issues.

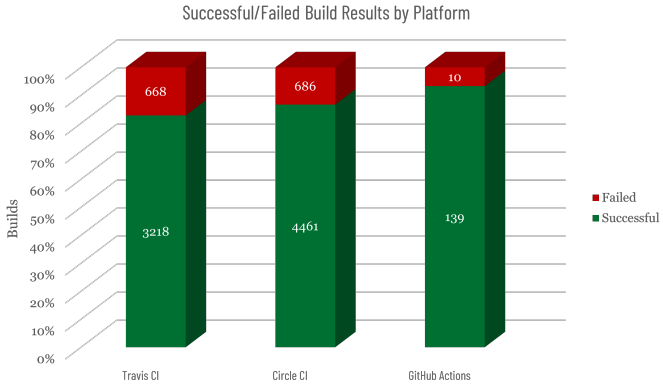


Fig. 7. Successful/Failed Build Results by Platform

**RQ2**: To investigate the failed builds further, a sample size of 300 was selected from the 1364 failed builds. The selection was done randomly to ensure that the sample is representative of the population. The sample size of 300 was determined

based on a confidence level of 95% and a margin of error of 5%. The confidence level of 95% means that if the same experiment is repeated 100 times, in 95 of those times, the results would fall within the given margin of error. The margin of error of 5% indicates that the results obtained from the sample of 300 failed builds can vary by a maximum of 5% from the actual population value. Finally, after selecting the sample of 300 failed builds, further analysis can be conducted manually to determine the causes of the failures. **Figure 8** shows a breakdown of the breakage types identified and their frequency.

The Breakage types can be defined as follows:

- **Version Incompatibility** - version of a dependency used in a project is not compatible with other dependencies or with the project itself
- **Source Code Errors** - where a dependency is used incorrectly or formatted improperly in the code, leading to errors or issues during the build process
- **Unused/Conflicting Dependencies** - dependencies that conflict with other dependencies such as having the same name but different version numbers
- **Vulnerable Dependencies** - dependencies that have known security vulnerabilities. These vulnerabilities can be exploited by attackers to gain unauthorized access to the system or data, or to cause other types of damage
- **Missing Dependencies** - where a required dependency is missing or has been removed from the project
- **Non-Dependency Related Breakages** - issues that are not directly related to dependencies, but still caused a broken build

It is notable that version incompatibility was by far the most common cause of dependency-related breakages, accounting for 239 instances in the sample. This highlights the importance of carefully managing dependencies and ensuring that they are compatible with each other and with the project being developed. Other notable causes of breakages included issues not related to dependencies (37 instances), source code changes (12 instances), and unused or conflicting dependencies (8 instances). These factors emphasize the importance of thorough testing and review of code changes, as well as careful management of dependencies to avoid issues that can cause breakages. In addition, the sample revealed that vulnerable dependencies were responsible for 1 instance of breakage, indicating the importance of regularly checking for and addressing security issues in dependencies. Missing or removed dependencies were identified as the cause of 3 breakages, highlighting the need to ensure that all necessary dependencies are included in the project and properly managed.

## VI. Threats to Validity

**Construct Validity** - In this analysis, there is a potential threat to construct validity in that not all commits that touched the package.json file and triggered a build might have actually caused the build breakage. To mitigate this threat, build logs were manually analyzed to confirm whether the commits were actually responsible for the breakage. This manual process may introduce errors or inconsistencies, and there is a risk that some build breakages may have been missed or misattributed. However, this was necessary to ensure the accuracy and validity of the analysis.

| Breakage Types | Number of Breakages |
|---|---|
| Version Incompatibility | 239 |
| Source Code Errors | 12 |
| Unused/Conflicting Dependencies | 8 |
| Missing Dependencies | 3 |
| Vulnerable Dependencies | 1 |
| Non-Dependency Related Breakages | 37 |

Fig. 8. Breakage Types Identified and their Frequency

**External Validity** - Another potential threat is the fact that only NPM was analyzed in this study. This means that the results may not be generalizable to other package managers. However, it should be noted that NPM is currently the largest software registry in the world and has more widespread adoption globally than other package managers, which makes it a suitable candidate for analyzing quality projects that are beneficial to the developer community on a global scale.

Overall, while there are potential threats to the construct and external validity of this analysis, steps were taken to mitigate these risks and the results are likely to be informative and useful for developers and project managers working with NPM-based software projects.

## VII. Conclusion

In conclusion, this analysis highlights the prevalence of dependency-related build breakages in NPM-based software projects. We found that 14.8% of builds were affected by dependency-related breakages, which can cause significant delays and disruptions in software development projects. Our results show that version incompatibility is the most common cause of such breakages, followed by incorrect formatting for using a dependency, and missing or removed dependencies. These findings underscore the importance of careful management and maintenance of software dependencies, and the need for developers and project managers to be vigilant in identifying and addressing potential breakages.

Moreover, our analysis provides insights into the types and frequency of dependency-related breakages that can occur in NPM-based projects. This information can help developers and project managers to better understand the challenges associated with dependency management and to develop more effective strategies for avoiding or mitigating breakages.

Overall, this study highlights the importance of ongoing monitoring and management of software dependencies in NPM-based projects. By addressing these issues proactively, developers and project managers can minimize the risk of build breakages and ensure that their software projects remain on track and on schedule.

## REFERENCES


[1] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and Propagation of NPM vulnerability fixes," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–24, 2021.

[2] S. M. Christian Macho and M. Pinzger, "Automatically Repairing Dependency-Related Build Breakage," *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1–12, 2018.

[3] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 1–33, 2017.

[4] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1–12, 2020.

[5] E. S. Suhaib Mujahib, Rabe Abdalkareem and S. McIntosh, "Using Others' Tests to Identify Breaking Updates," *17th International Conference on Mining Software Repositories (MSR '20)*, pp. 1–11, 2020.

[6] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. Wiese, "I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages," *ACM Transactions on Software Engineering and Methodology*, pp. 1–25, 2023.