CS846 Final Report: Tracking Dependency Updates & Security: Do Bots Make a Difference?

Jaffer Iqbal
j2iqbal@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Eimaan Saqib
e2saqib@uwaterloo.ca
University of Waterloo
Waterloo, Canada

ACM Reference Format:
Jaffer Iqbal and Eimaan Saqib. 2025. CS846 Final Report: Tracking Dependency Updates & Security: Do Bots Make a Difference?. In . ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 ABSTRACT

Modern software systems rely heavily on open-source dependencies, which introduces challenges in managing security, freshness, and maintenance at scale. Dependency management bots like Dependabot aim to automate updates, but their real-world impact remains underexplored. This study investigates how projects on Maven Central manage dependency updates and respond to vulnerabilities, introducing metrics such as Dependency Freshness and Vulnerability Exposure Windows. By analyzing over 38,000 artifacts, we evaluate the variation of these metrics across projects of different sizes, popularity, and types. Our findings reveal that while Dependabot adoption significantly improves dependency freshness, its effect on reducing vulnerability exposure windows is statistically insignificant. These results offer empirical insights into the strengths and limitations of automated dependency management practices.

2 INTRODUCTION

Modern software projects heavily use open-source dependencies. But managing these dependencies in large projects comes with risks that include security risks, technical debt, compliance issues, etc. Vulnerabilities in dependencies are a significant cause of software breaches. For example, the 2021 Log4j vulnerability (CVE-2021-44228) exposed millions of systems globally and demonstrated the high risk of unpatched vulnerabilities in the software ecosystem [13]. A 2024 report by Synopsys found that 84% of open-source codebases still contained at least one open-source vulnerability, while 74% included high-risk vulnerabilities, marking a 54% increase from 2022 [2, p. 6]. This highlights the ongoing need for improved dependency management and maintenance practices.

To address these challenges, automated dependency management tools (often referred to as *dependency management bots*) have been introduced to automate the process of updating dependencies. Several tools provide automated dependency management services, including GitHub's Dependabot [5], Renovate Bot [9], Snyk [14], and Depfu [1], which have been developed to help maintain dependency hygiene, security, and software updates. While these tools are used across open-source projects, their effectiveness in real-world ecosystems such as Maven Central (the largest repository of Java libraries) remains unclear. To make informed investment and adoption decisions, organizations need empirical evidence of how these tools impact software security, maintenance practices, and dependency stability within large-scale open-source projects.

This study seeks to bridge this gap by evaluating the effectiveness of Dependabot's dependency management practices within the Maven ecosystem.

Our research objective is divided into three research questions. We define *Dependency Freshness* as the time elapsed between the adoption of a dependency by a project and the latest available release of that dependency, and *Vulnerability Exposure Windows* as windows of time in which an artifact is vulnerable - these windows are divided into 3 sub-types as defined in Table 1.

(1) **RQ-1:** What is the average Dependency Freshness in projects, and how does it vary across projects of different sizes, popularity, and type?
(2) **RQ-2:** What are the average Vulnerability Exposure Windows in projects, and how do they vary across projects of different sizes, popularity, and type?
(3) **RQ-3:** Does the adoption of dependency management bots correlate with reduced Dependency Freshness Vulnerability Exposure Windows?

Our findings provide practical insights into how these tools perform in the real-world software. Specifically, we highlight their impact on maintenance practices across diverse Java projects on Maven Central. In doing so, we advocate for the informed and strategic adoption of dependency management bots to improve the dependency freshness in large-scale software systems.

3 METHODOLOGY

3.1 Dataset and Tools

This study makes the use of the Goblin framework [8], a Neo4J-based dependency graph of Maven Central artifacts and releases. This graph (`with_metrics_goblin_maven_30_08_24.dump`[7]) is a snapshot of the dependency graph dated on 30 Aug 2024, comprising of 15,117,217 nodes (658,078 artifacts and 14,459,139 releases) that can be queried via Cypher [10]. The dataset also contains 44,035,495 *AddedValue* nodes with additional information like CVEs affecting a release (sourced from a copy of the OSV dataset [12]) and precomputed metrics like *popularity* and *freshness*. We find 77,393 releases (belonging to 1,411 artifacts) containing at-least one CVE, and 197,186 unique artifacts depending on these vulnerable releases in their lifetimes. We use the GitHub API [3] to find open-sourced projects using Dependabot in their pipeline. Finally, we use Python libraries like `Pandas`, `Scipy`, and `Plotly` to perform statistical analysis and visualize trends in dependency management behavior.

3.2 Gathering Project Metrics for Stratified Analysis

We stratify Maven artifacts (i.e. projects) based on their size, popularity and type. We consider a project's Number of Dependencies (*numDependencies*) and Lines of Code (*LoC*) as indicative of its size as they reflect a project's complexity and development scale. To measure a project's popularity, we look at its Number of Dependents (*numDependents*) and GitHub , as they reflect ecosystem reliance and community interest. Finally, we find the project's type (*type*), which is an indicator of its intended usage in the Maven Ecosystem. The classification of project types is determined using a combination of heuristic checks and external data sources. Initially, the artifact name is checked for keywords; the project is classified as a framework or application. If the number of dependencies is greater than 1000, the project is classified as a framework, while fewer than 10 dependencies result in classification as an application. If the project has a GitHub repository, the repository's topics are fetched to check for keywords such as "framework" or "sdk" (indicating a framework) or "cli" or "app" (indicating an application). The project's POM file is also checked for packaging information and plugins; if specific plugins like spring-boot-maven-plugin or wildfly-maven-plugin are found, it is classified as an application. If none of these conditions are met, the project is classified as a library.

Finding numDependencies and numDependents was straightforward for all 658,078 artifacts by querying the dependency graph. However, the remaining metrics (LoC, GitHub stars and type) had to be fetched by making muliple queries to the GitHub API per project. Since the GitHub API enforces strict rate limits [4], it was not feasible to retrieve these metrics for all 658,078 artifacts. We were able to gather these metrics for a set of 38,794 randomly chosen artifacts. To ensure consistent comparisons across all stratification dimensions, we limit our analysis to this reduced set of 38,794 artifacts — despite having numDependencies and numDependents available for the full set of 658,078 artifacts.

3.3 Identifying Dependency Freshness

Dependency Freshness is defined as the time elapsed between the adoption of a dependency by a project and the latest available release of that dependency. In other words, it quantifies how out-of-date a project's dependency is relative to its most recent version. This was computed by querying the dependency graph. To compute, we first identify each project's release that includes a specific dependency, then determine the most recent release timestamp of that dependency from the broader ecosystem. The freshness for that dependency is calculated as the difference (in days) between the project's adoption timestamp and the dependency's latest release timestamp. By aggregating these freshness values across all releases of a project, we compute summary statistics (average, median, maximum, and minimum freshness) which provide insight into how proactively projects update their dependencies and mitigate potential risks associated with outdated components.

Table 1: Summary of Vulnerability Exposure Windows and Their Descriptions.

Exposure Window Type	Description
True Exposure Window	(*exposureEndTime* - *exposureStartTime*), quantifies the total duration an artifact remains vulnerable.
Known Exposure Window	(*exposureEndTime* - *cvePublishTime*), captures the period during which the vulnerability was publicly disclosed yet unaddressed.
Patch Lag Window	(*exposureEndTime* - cveFixTime), reflects the delay between the fix release and the project's adoption of a non-vulnerable version.

3.4 Identifying Vulnerability Exposure Windows

Vulnerability Exposure Windows are defined as windows of time in which an artifact is vulnerable because it depends on a dependency that is known to be vulnerable. For an artifact A that depends on artifact B containing known vulnerable releases, we use Cypher queries to find the oldest release of artifact A (*artifactA_Release0*) which depends on a vulnerable release of artifact B and record its timestamp as the Starting Time of Exposure to Vulnerability (*exposureStartTime*). Next, we find the oldest release of artifact A (following *artifactA_Release0*) which no longer depends on a vulnerable release of artifact B and record its timestamp as the Ending Time of Exposure to Vulnerability (*exposureEndTime*). We query the OSV API [11] to find the time when the specific CVE was published (*cvePublishTime*) and the time when a fix was released for that CVE(*cveFixTime*). We find that fixed versions were not published for 996/1411 (29.4%) of afflicted artifacts in the dependency graph.

We divide the vulnerability exposure windows into three sub-types as shown in Table 1. The *True Exposure Window* indicates the total risk period during which a project remains vulnerable, serving as a baseline measure of exposure. The *Known Exposure Window* highlights the period when the vulnerability has been publicly disclosed but remains unaddressed, emphasizing the window of external pressure and potential exploitation. Finally, the *Patch Lag Window* reflects the delay between the availability of a fix and its adoption by the project, serving as an indicator of a project's responsiveness and efficiency in mitigating vulnerabilities. Note that an artifact A may depend on multiple vulnerable artifacts (e.g., B and C), each with its own set of exposure windows. To obtain a single value for each exposure window sub-type for artifact A, we average the corresponding sub-type windows across all its vulnerable dependencies. For example, the True Exposure Window for A is calculated by averaging the True Exposure Windows from dependencies B and C, and similarly for the Known Exposure and Patch Lag Windows.

3.5 Identifying Dependency Freshness and Vulnerability Exposure Windows in Control and Test Group

Once the project metrics, dependency freshness and vulnerability exposure windows have been computed for our reduced set of randomly chosen 38,794 artifacts, we partition it into two groups: artifacts that do not use Dependabot for automated dependency management (36,292), and artifacts that do (2,502). To perform a direct comparison between these two groups and establish their statistical significance, we take a random sample of the first group to match the size of the second group. We had initially attempted to identify bot usage across 4 dependency management bots (Dependabot, Renovate Bot, Synk and Depu), but found insufficient representation of bot usage other than Dependabot. Since [15] confirms that Dependabot is the most used bot for Github projects, we exclusively focus on it for subsequent analysis.

Projects that use Dependabot for automated dependency management contain a config file (`dependabot.yml`) located in the (`./GitHub`) directory of a project [6]. We use the GitHub API's code search functionality to locate this config file in the projects' repositories. If this file was found, then the project is classified as using Dependabot for automated dependency management.

4 RESULTS AND DISCUSSION

4.1 RQ1

What is the average Dependency Freshness in projects, and how does it vary across projects of different sizes, popularity, and type?

Our analysis of dependency freshness indicates that extreme outliers constitute less than 3% of the data. Because these outliers are symmetrically distributed around the center, the mean and median dependency freshness values are very similar. For clarity and robustness of our analysis, we have removed these extreme values. Overall, the distribution of dependency freshness is approximately uniform with a slight right skew, suggesting a modest tendency towards longer exposure durations (Fig 1).

When comparing dependency freshness across project types, we observe distinct patterns. Projects classified as applications tend to have a sharp, well-defined peak in their freshness distribution, indicating a more consistent update behavior. In contrast, libraries and frameworks show a flatter distribution, which may reflect more variable dependency management practices in these project categories (Fig 2).

Our correlation analysis reveals that projects with higher dependent counts and larger codebases (as measured by lines of code) tend to have lower dependency freshness – that is, their dependencies are updated more frequently (Fig 3). Conversely, projects with more GitHub stars show higher dependency freshness. This suggests that larger, actively maintained projects are more proactive in updating dependencies, whereas popular projects might exhibit longer intervals between dependency updates, possibly due to increased complexity or stability considerations.

To further explore these relationships, we stratified the data by project type, project size (lines of code), and project popularity (GitHub stars). The stratified analysis indicates that projects with smaller codebases and lower popularity tend to exhibit a slightly more right-skewed and flatter freshness distribution (Fig ??). This implies greater variability in dependency update practices among these projects.

Finally, a regression analysis was conducted to predict average dependency freshness from project size, popularity, and dependent count. The results show that GitHub stars have a statistically significant positive association with dependency freshness, while both lines of code and dependent count are negatively correlated with freshness ($p < 0.05$ for all predictors). These findings suggest that, after controlling for other factors, projects that are larger and have more dependents tend to update their dependencies more frequently, whereas more popular projects are associated with longer dependency freshness intervals.

Overall, our results imply that proactive dependency management – characterized by lower dependency freshness – is more common in projects with larger codebases and higher dependent counts. In contrast, popular projects (with higher GitHub stars) tend to update dependencies less frequently.

4.2 RQ2

What are the average Vulnerability Exposure Windows in projects, and how do they vary across projects of different sizes, popularity, and type?

The analysis of vulnerability exposure windows reveals several important findings. Outliers in the exposure window data are mostly towards the higher end of the data, suggesting that the data is not heavily skewed by extreme low values. There is a notable difference between the mean and median exposure windows, with the mean being significantly larger, which indicates a right-skewed distribution. This right-skewness suggests that while most projects have relatively short exposure windows, a smaller subset has significantly longer exposure periods. The percentage of outliers is also considerable, reinforcing the presence of a few projects with extended vulnerability exposure (Fig 4).

The distribution of vulnerability exposure windows shows distinct patterns across project types. Framework projects tend to have a flatter distribution, indicating that these projects exhibit a more uniform exposure window length. In contrast, applications and libraries display more peaked distributions, suggesting that these project types may have more concentrated vulnerability exposure periods (Fig 5).

In terms of correlations, we find that GitHub stars and dependent count are negatively correlated with the vulnerability exposure window, meaning that projects with more dependents and higher popularity tend to have shorter exposure windows (Fig 6). This may indicate that popular or widely used projects are quicker to patch vulnerabilities, possibly due to greater community attention and pressure. On the other hand, lines of code show a positive correlation with the exposure window, suggesting that larger projects may take longer to address vulnerabilities, possibly due to their complexity and scale.

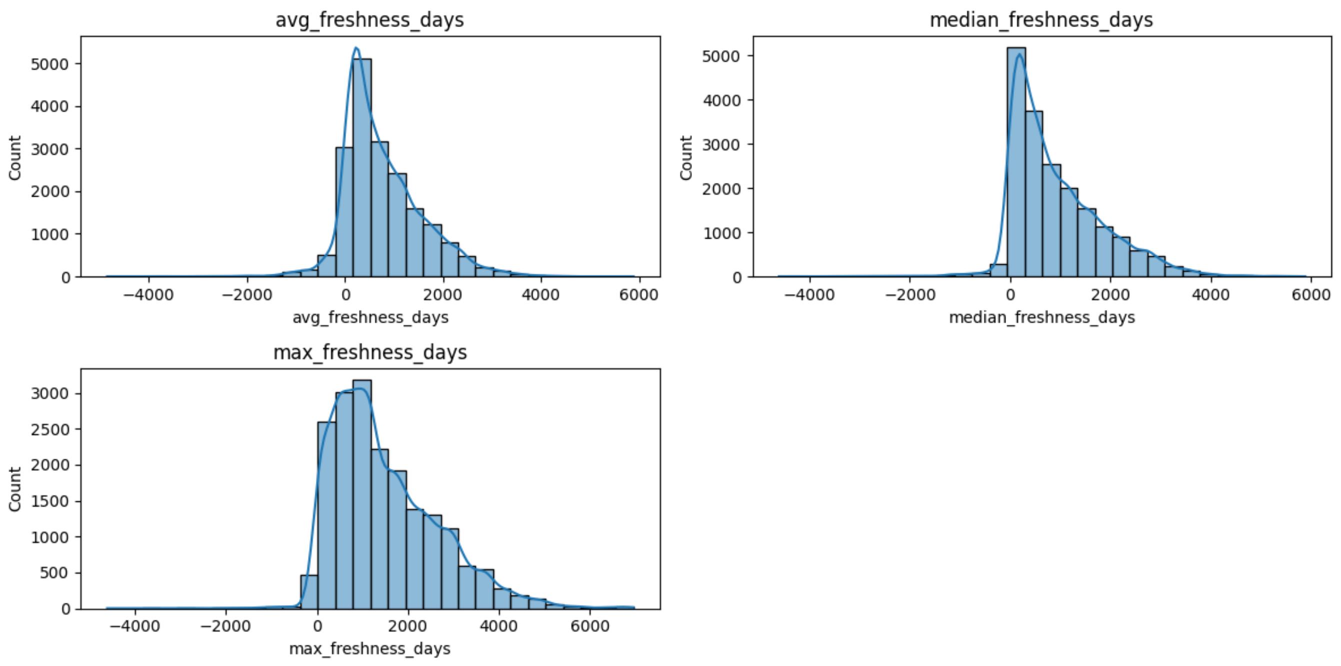


Figure 1: Distribution of Dependency Freshness

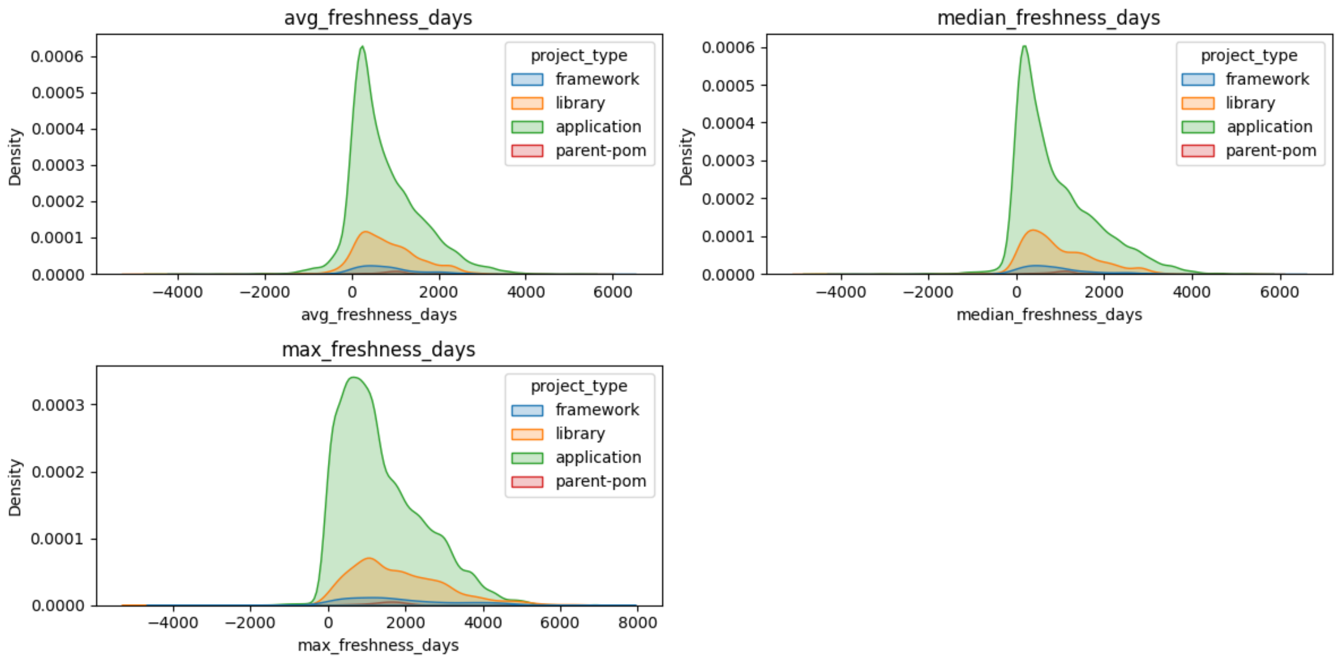


Figure 2: Dependency Freshness by Project Type

We further stratified the data by project size (lines of code) and popularity (GitHub stars) to observe the differences in exposure windows across different project categories (Fig ??). Both size and popularity distributions exhibit a second slight peak, suggesting that some projects experience a secondary increase in vulnerability exposure at a slightly longer window. For the middle popularity and size bins, the second peak appears at lower exposure windows,

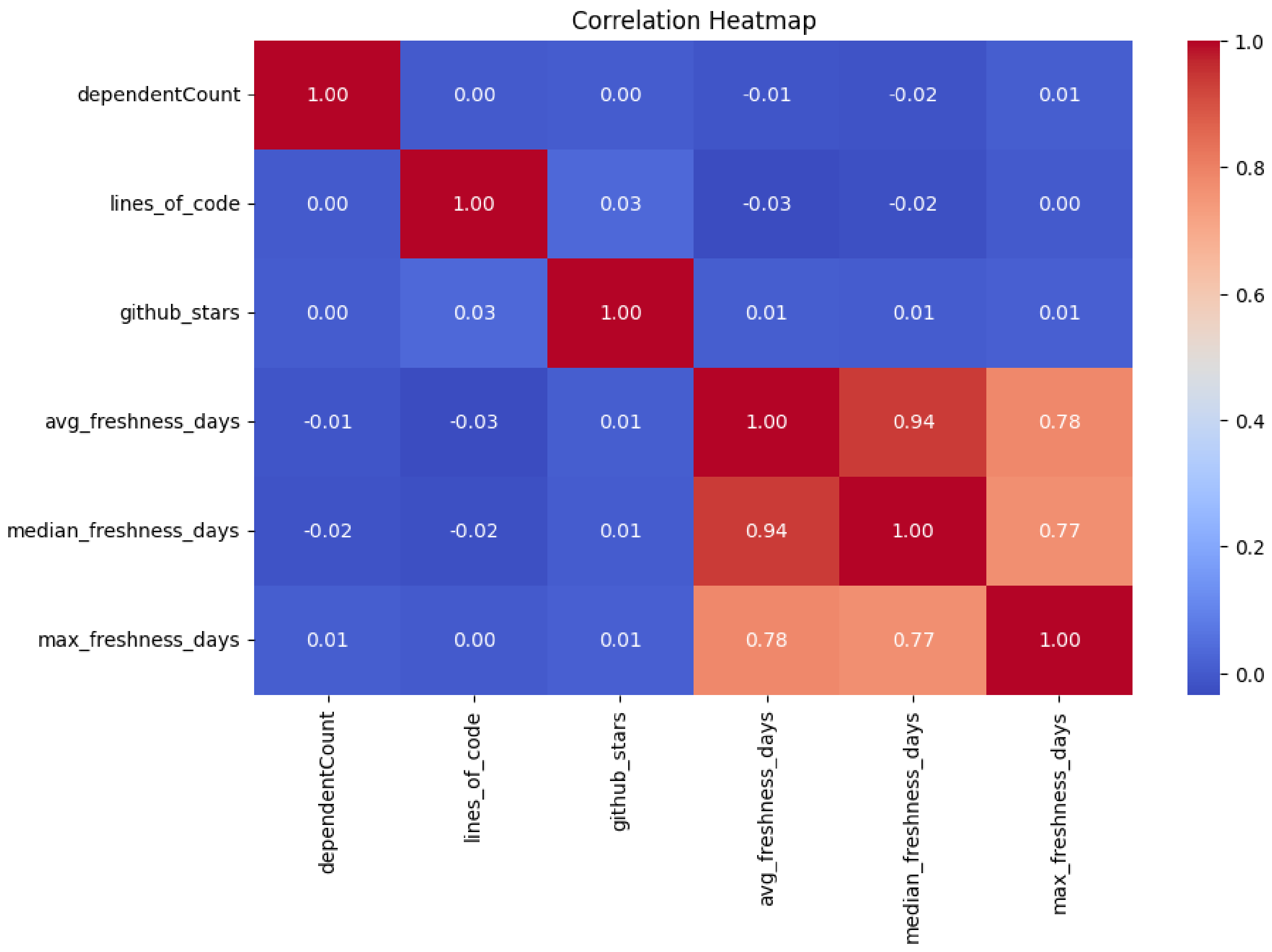


Figure 3: Correlation Heatmap

suggesting that these projects may have a more rapid escalation of exposure after initial delays.

Finally, regression analysis was conducted to model the vulnerability exposure window based on project size, popularity, and dependent count. The regression results confirm the correlations observed in the earlier analysis: higher GitHub stars and dependent count are associated with shorter exposure windows, while larger projects (in terms of lines of code) tend to have longer exposure windows. All predictors have significant p-values ($p < 0.05$), reinforcing the importance of these factors in explaining the variability in vulnerability exposure windows.

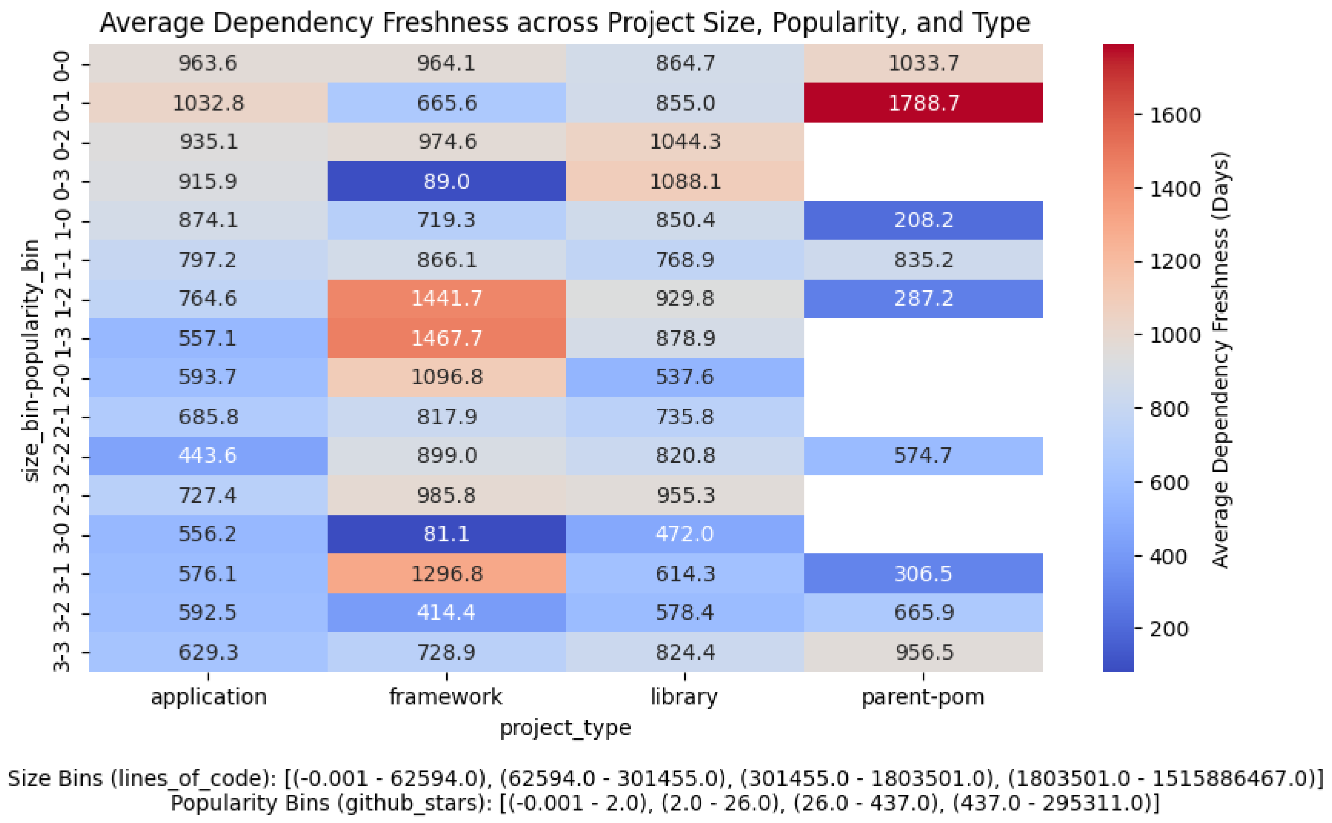
In summary, our findings suggest that popular projects with more dependents tend to have shorter vulnerability exposure windows, likely due to more active maintenance and community attention. On the other hand, larger projects with more complex codebases appear to have longer exposure windows.

4.3 RQ3

Does the adoption of dependency management bots correlate with reduced Dependency Freshness Vulnerability Exposure Windows?

We examined the relationship between the adoption of the dependency management tool, Dependabot, and the vulnerability exposure windows as well as the dependency freshness across the treatment (no Dependabot) and control (Dependabot used) groups. The analysis was conducted using a series of statistical tests, including t-tests, U-tests, and regression analysis.

For the vulnerability exposure window, we found no statistically significant difference between the treatment and control groups. The t-test yielded a t-statistic of -1.335 and a p-value of 0.182, while the U-test produced a U-statistic of 188886.5 with a p-value of 0.166. These results indicate that the adoption of Dependabot does not significantly reduce the vulnerability exposure window. Furthermore, the regression analysis revealed that the coefficient for the variable dependabot_used was -45.11, with a p-value of 0.295, suggesting



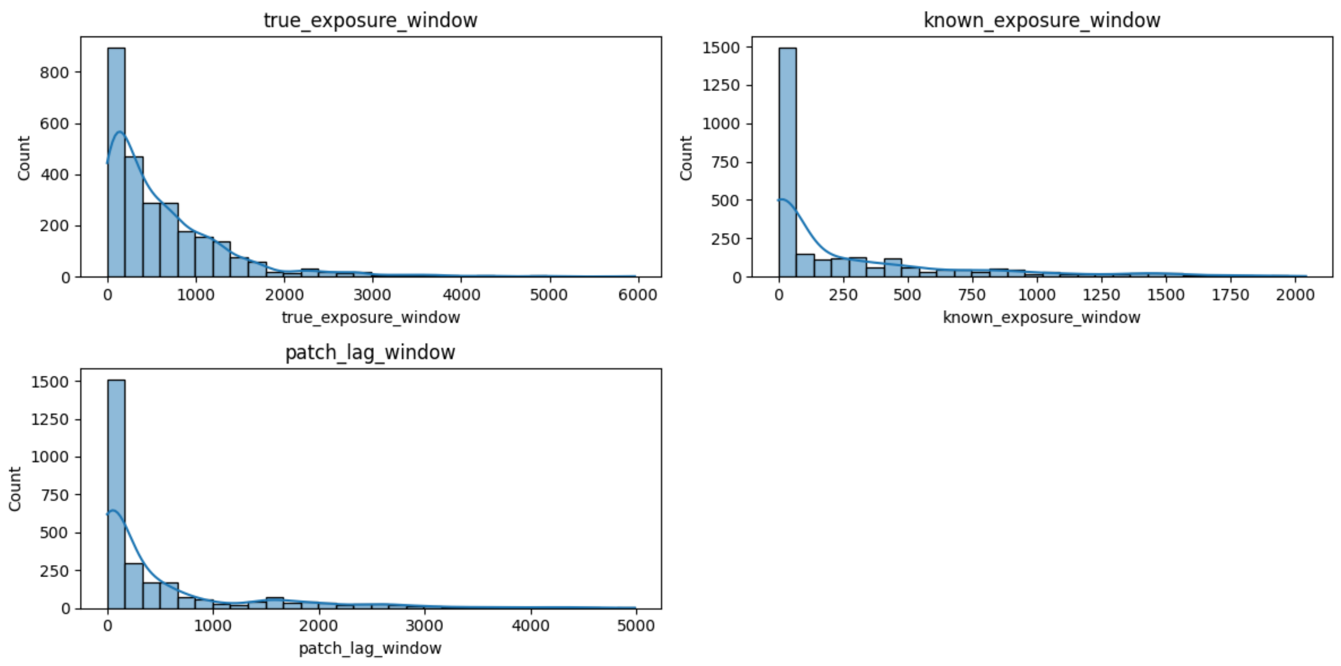


Figure 4: Distribution for Exposure Windows

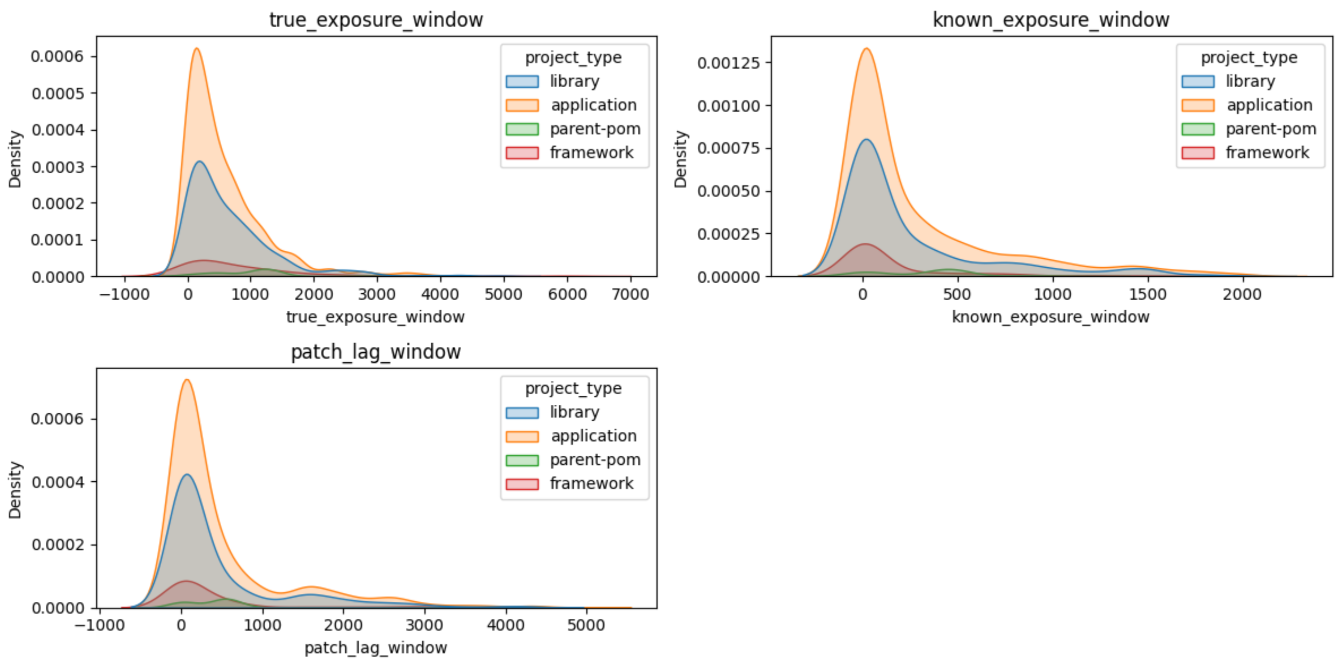


Figure 5: Exposure Windows by Project Type

no significant correlation between the use of Dependabot and the length of the vulnerability exposure window.

However, when examining the distribution of vulnerability exposure windows, we observed that the control group (artifacts using Dependabot) had lower quartile values and fewer extreme outliers compared to the treatment group, which suggests that while the difference in means may not be significant, there are potentially fewer extreme vulnerabilities in projects utilizing Dependabot (Fig 7).

On the other hand, for dependency freshness, the results were more revealing. The t-test for the difference in average dependency freshness between the treatment and control groups yielded a t-statistic of 5.93 and a highly significant p-value of 3.3e-09, indicating a clear and significant difference. The U-test also returned a significant p-value of 9.47e-09. The regression results further corroborated these findings, showing that the adoption of Dependabot is associated with an increase in dependency freshness, with a coefficient of 129.65 (p-value = 0.000).

The regression analysis also revealed that the number of lines of code had a negative correlation with dependency freshness (coefficient: -3.05e-07, p-value < 0.001), while GitHub stars had a positive correlation (coefficient: 0.0046, p-value = 0.002).

Interestingly, although projects using Dependabot exhibited higher dependency freshness on average, the box plot for the control group (treatment group without Dependabot) showed a slightly wider distribution compared to the treatment group, indicating greater variability in the dependency freshness values for projects that did not use Dependabot (Fig 8).

In summary, our analysis indicates that while the adoption of Dependabot does not have a statistically significant effect on reducing vulnerability exposure windows, it is associated with significantly improved dependency freshness, with projects using Dependabot maintaining fresher dependencies. These findings suggest that while Dependabot may not immediately impact the speed at which vulnerabilities are patched, it could still help in maintaining the overall freshness of dependencies, which could indirectly contribute to more secure and up-to-date project environments.

5 THREATS TO VALIDITY AND FUTURE WORKS

Repositories containing a `.yml` configuration file (like `dependabot.yml`) may not actively use the dependency management bot. This can occur due to several reasons, such as mis-configured settings, disabled automation, archived or inactive repositories, or a lack of dependencies matching the update criteria. To mitigate this, we can verify active bot usage by checking for pull requests authored by the bot, analyzing commit histories for dependency updates, and querying repository settings to confirm if the bot is enabled. This ensures that only repositories with demonstrable bot activity are considered. However, this was not possible in the current timeline of the project considering the rate limits imposed on the GitHub API. This limitation can be addressed in future works by incorporating bot activity verification through pull request analysis and commit history mining.

There is a potential risk of selection bias in this study, as projects that adopt dependency management bots, such as Dependabot, may inherently prioritize security and better dependency management

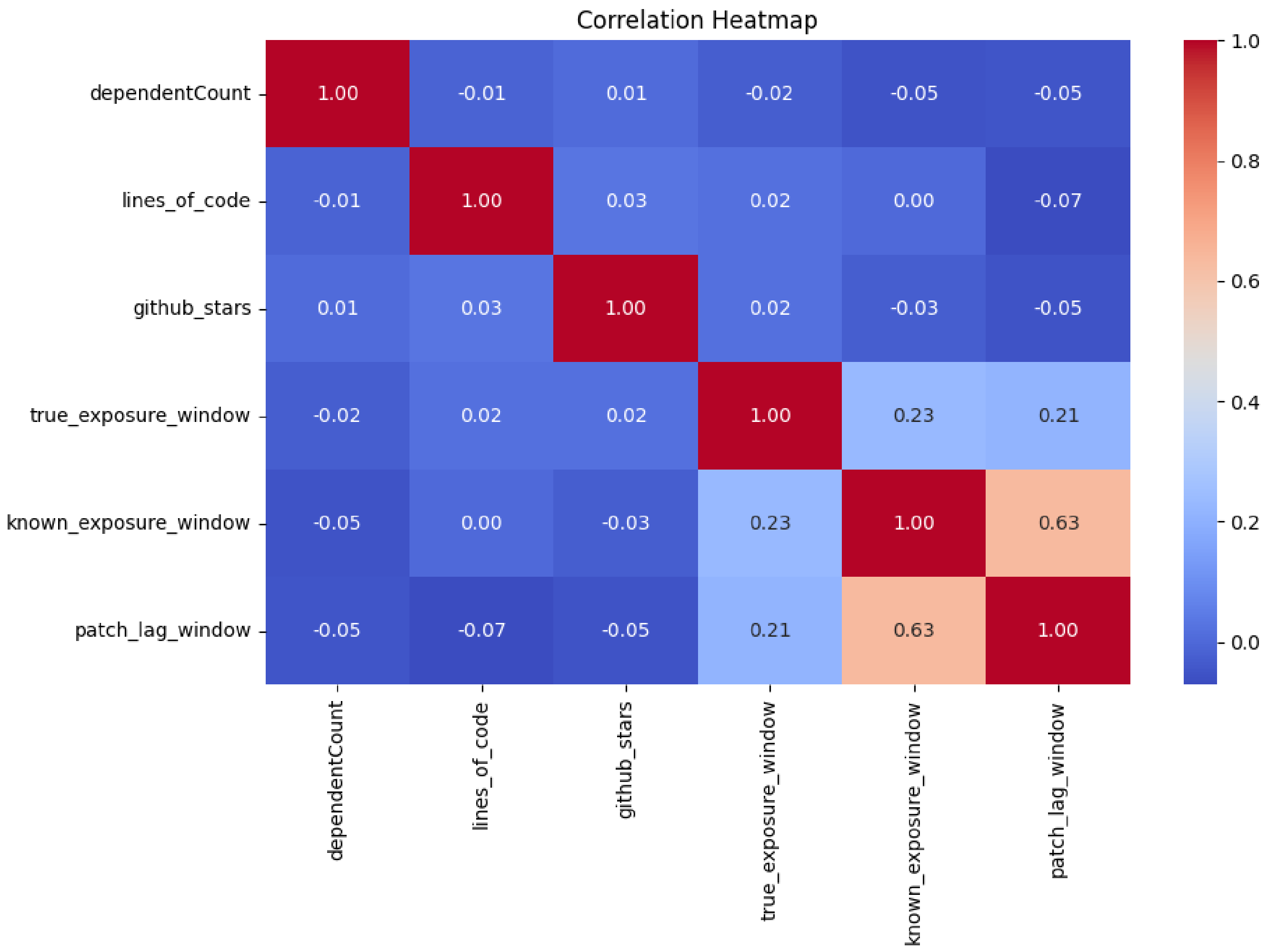


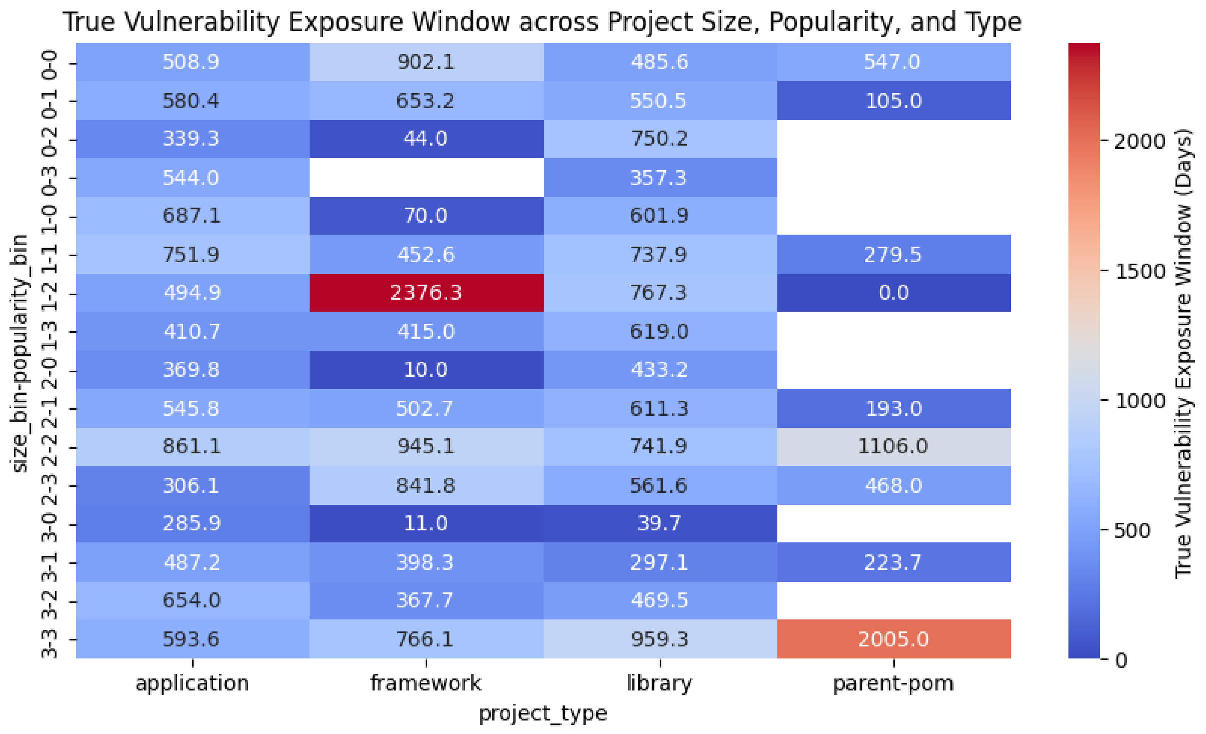
Figure 6: Correlation Heatmap

practices compared to other projects. This difference in prioritization could introduce a skew in the results, as projects with a greater focus on security and maintenance might naturally have improved dependency freshness and vulnerability management, regardless of the tool used. Also, it's important to acknowledge the issue of causal ambiguity; while a correlation between the use of dependency management bots and improved metrics (such as reduced vulnerability exposure windows or improved dependency freshness) may exist, this does not necessarily imply a causal relationship.

To mitigate the risk of selection bias, future studies could implement more robust control groups by matching projects based on similar size, popularity, and domain. This would help ensure that the observed differences in outcomes are more likely to be due to the intervention (i.e., the use of dependency management bots) rather than pre-existing differences between the groups. Furthermore, a longitudinal study could provide more insight into the long-term impact of these tools. By isolating the effect of the tool over time and tracking the same projects before and after the adoption of dependency management bots, future research could establish a stronger correlation and potentially uncover a causal relationship between the use of these tools and improved dependency management outcomes.

6 RELATED WORKS

A comprehensive exploratory study on GitHub's Dependabot [6], examines both quantitative and qualitative aspects of its performance. Their mixed-methods approach combined exploratory data analysis with developer surveys to evaluate Dependabot's effectiveness in keeping dependencies up-to-date. Their findings revealed that projects generally reduced technical lag after adopting Dependabot, with developers showing high receptivity to automatically generated pull requests. However, they also identified limitations, including inadequate compatibility scores for reducing update suspicion and a tendency for developers to configure Dependabot to minimize notification volume. Notably, 11.3% of projects in their



Size Bins (lines_of_code): [(-0.001 - 125994.0), (125994.0 - 870243.0), (870243.0 - 4225326.0), (4225326.0 - 1515886467.0)]
Popularity Bins (github_stars): [(-0.001 - 6.0), (6.0 - 53.0), (53.0 - 823.0), (823.0 - 116860.0)]

study eventually deprecated Dependabot in favor of alternative solutions. Our work extends this by focusing specifically on the Maven ecosystem, introducing novel metrics like Dependency Freshness and Vulnerability Exposure Windows, and empirically evaluating how these metrics vary across projects with different characteristics.

6.1 Conclusion

In this study, we investigate the impact of automated dependency management (specifically GitHub's Dependabot) on real-world Java projects in the Maven ecosystem. We introduce two novel metrics, Dependency Freshness and Vulnerability Exposure Windows, to quantify how promptly projects update dependencies and respond to vulnerabilities. By analyzing a curated dataset of over 38,000 artifacts, we stratify projects by size, popularity, and type, and compare bot-using projects against non-users. Our findings show that Dependabot usage significantly improves dependency freshness, but does not have a statistically significant impact on reducing vulnerability exposure windows. These insights emphasize the strengths and limitations of automated tools in managing dependencies at scale.

REFERENCES

[1] Depfu. [n. d.]. Depfu. https://depfu.com/.
[2] Synopsys Black Duck. 2024. *Open Source Security and Risk Analysis Report.* Technical Report. Black Duck by Synopsys. 18 pages. https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html Accessed: 2024-02-21.
[3] GitHub. [n. d.]. Github REST Api Documentation. https://docs.github.com/en/rest?apiVersion=2022-11-28
[4] GitHub. [n. d.]. GitHubAPIRateLimit. https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28.
[5] Github. [n. d.]. GitHub's Dependabot. https://github.com/dependabot.
[6] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4004–4022.
[7] Damien Jaime. [n. d.]. withMetricsDataset. https://zenodo.org/records/13734581/.
[8] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2024. Goblin: A framework for enriching and querying the maven central dependency graph. In *Proceedings of the 21st International Conference on Mining Software Repositories.* 37–41.
[9] Michael Kriese. [n. d.]. Renovate Bot. https://github.com/renovatebot.
[10] Neo4j. [n. d.]. Cypher Tutorial. https://neo4j.com/docs/cypher-manual/current/queries/basic/.
[11] OSV. [n. d.]. OSV API. https://osv.dev/use-the-api.
[12] OSV. [n. d.]. OSV Website. https://osv.dev/.
[13] Red Hat Customer Portal. [n. d.]. 2021 Log4j vulnerability (CVE-2021- 44228. https://access.redhat.com/security/cve/cve-2021-44228.
[14] Synk. [n. d.]. Synk Bot. https://github.com/snyk-bot.
[15] Marvin Wyrich, Raoul Ghit, Tobias Haller, and Christian Müller. 2021. Bots don't mind waiting, do they? Comparing the interaction with automatically and manually created pull requests. In *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE).* IEEE, 6–10.

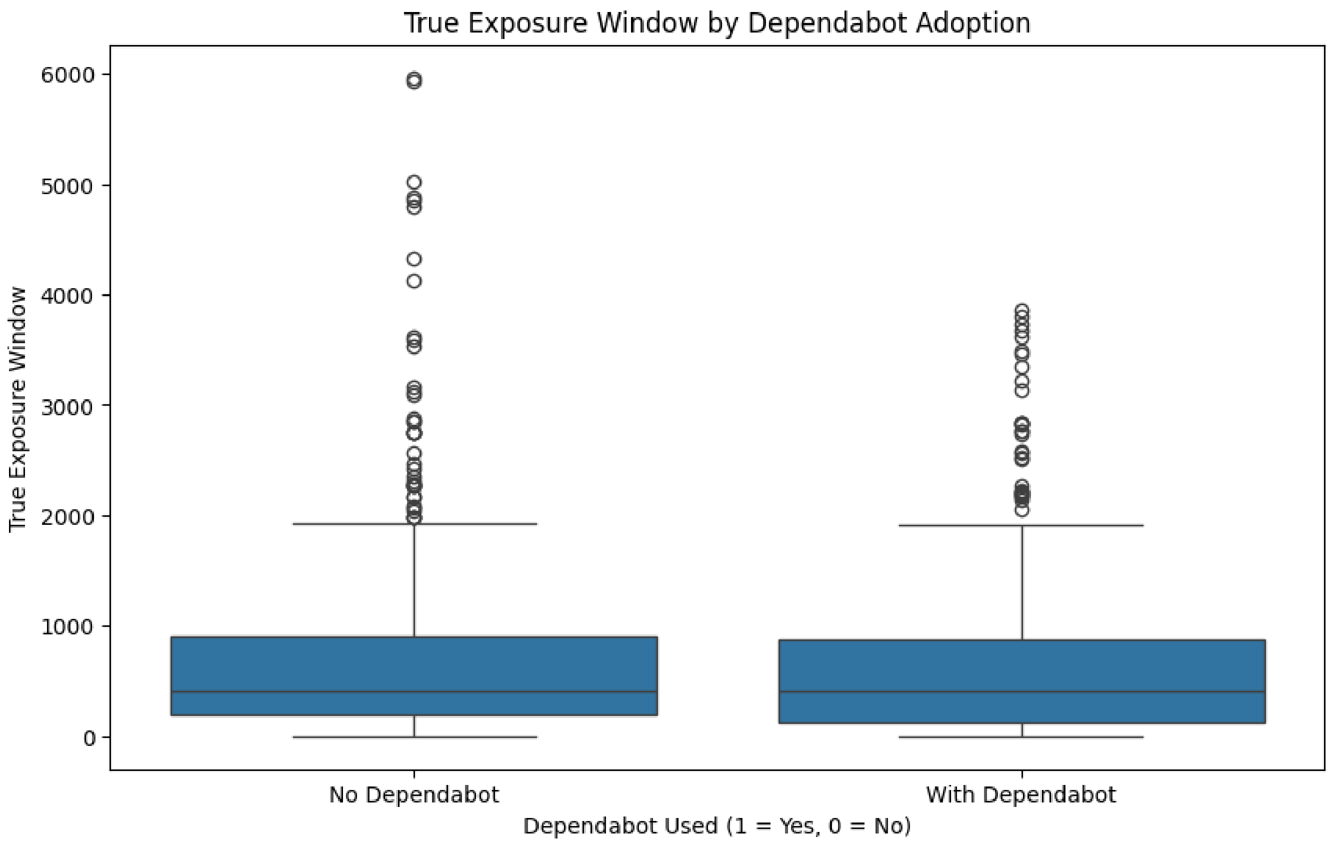


Figure 7: Exposure Windows with Dependabot Adoption

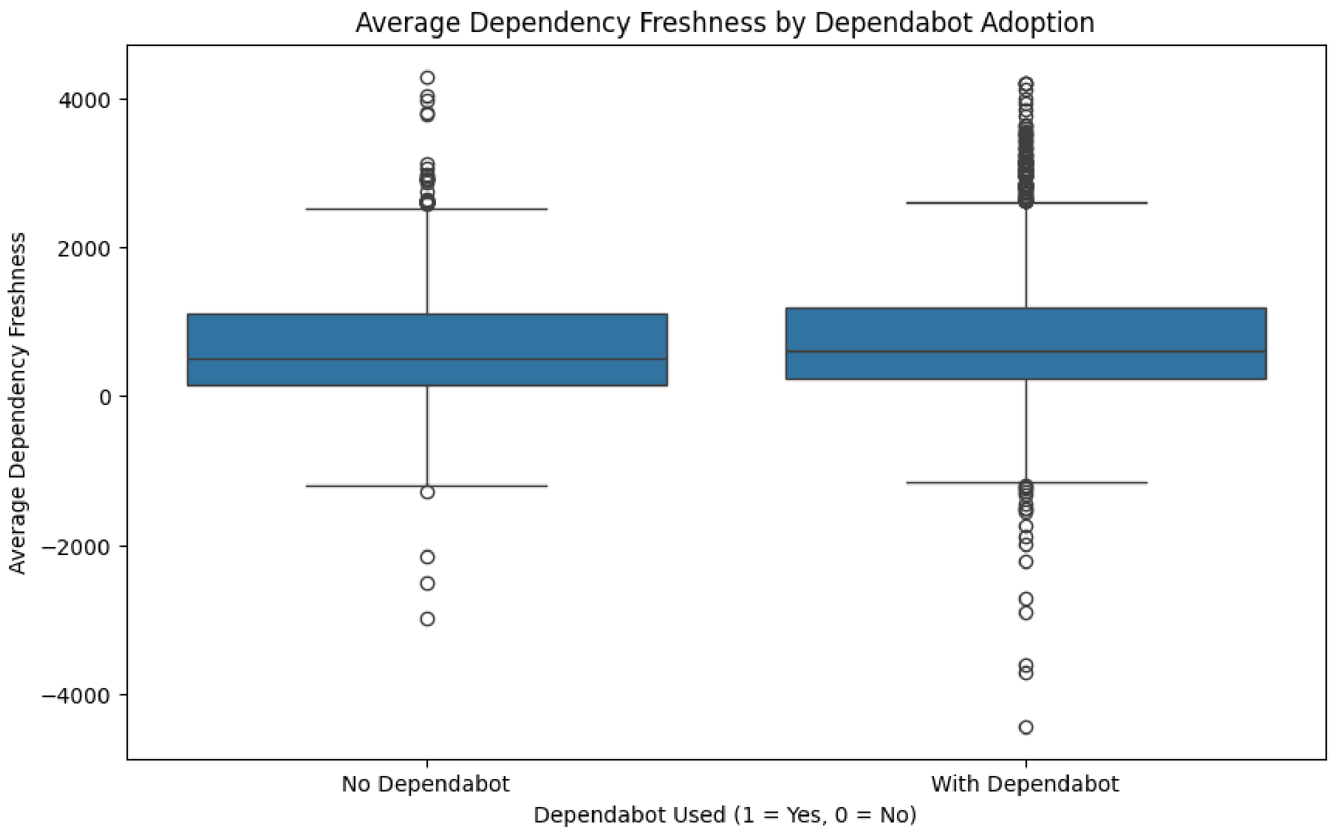


Figure 8: Dependency Freshness with Dependabot Adoption