Tracking Dependencies and Release Frequency in the Maven Architecture using Neo4j and Goblin Weaver

Daniel Pang
University of Waterloo
Waterloo, Ontario
d3pang@uwaterloo.ca

Ahmed El Shatshat
University of Waterloo
Waterloo, Ontario
ar2elsha@uwaterloo.ca

Abstract

System architecture plays a pivotal role in software engineering, influencing maintainability, security, and release cadence. One key aspect of architectural evolution is dependency management, where software artifacts may either accumulate or reduce dependencies over time. To assess how dependency counts evolve over time, we analyzed library dependency structures within the Maven Central build system using data from the MSR 2025 Mining Challenge. Our study investigates three research questions: (1) whether Maven libraries exhibit a trend towards fewer dependencies, (2) whether dependency trends correlate with software release speed, and (3) whether increased release frequency reduces security vulnerabilities. Our findings reveal a general increase in dependency counts across Maven libraries, with popular libraries exhibiting more nuanced patterns—often experiencing dependency bloat before later reductions. Due to dataset constraints, a comprehensive analysis of security vulnerabilities remains an open challenge. These results provide insights into the evolution of software ecosystems and suggest further areas of research into balancing dependency management with release velocity and security considerations.

Keywords

Maven, Neo4J, Cypher, Dependency Tracking, Goblin Weaver

ACM Reference Format:
Daniel Pang and Ahmed El Shatshat. 2025. Tracking Dependencies and Release Frequency in the Maven Architecture using Neo4j and Goblin Weaver. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

System architecture is a foundational part of software engineering, as the structure of a system has an impact on near every facet of the system. As a system grows in functionality and robustness, it often needs to increase its dependencies; this can be for a number of reasons, including achieving such new functionality, or addressing security issues that caused vulnerabilities in the system. For just as many reasons, a system may need to reduce its number of dependencies, as it may be needlessly bloating the system architecture, or introducing new security vulnerabilities if vulnerabilities are found.

However, it is generally believed that best practice is to keep your system artifacts with as few dependencies as possible. A system with as few dependencies as possible is thought to be far easier to manage, as the space for where bugs can appear or where version issues may manifest is reduced. It follows that, theoretically, system architecture should trend naturally downwards with time.

In practice, there are countless circumstantial factors that may prevent this from being the case. To gain greater insight on how dependency counts evolves over time, we analyzed libraries within the Maven Central build system.

1.1 Motivation

The Maven Central build system is the dataset for the MSR 2025 Mining Challenge [4]. Given the data was chosen and facilitated by a credible organization, it serves as a good resource by which to perform such an analysis. Maven itself is a 20 year old project that has been documented extensively; thus, there is a lot of data to peruse across its many years of being maintained.

As mentioned before, there is an assumption that dependency counts for libraries within a system should decrease over time, with the reasoning being that a smaller architecture is easier to manage. To that end, if we do see that dependency counts within Maven libraries decreasing, our next question is what effect this has had on release speeds, if any.

The dataset includes additional nodes that contain security vulnerability information, if a library or associated dependency has been flagged as having a security vulnerability. If the libraries trend towards fewer dependencies, the expectation is that there would be fewer security vulnerabilities overall. Furthermore, if there is an increase in release frequency, it should also follow that security vulnerabilities are being addressed more expediently.

1.2 Research Questions

To assess the veracity of our hypothesis, we formed these three research questions:

- RQ1: Has Maven trended towards software architecture that has fewer dependencies over the years?
- RQ2: If Maven has trended towards fewer dependencies, has this resulted in faster software lifecycles?
- RQ3: Has faster software life-cycles reduced security vulnerabilities in Maven?

The structure of the report is as follows. Section 2 will discuss methodology, including more details on the dataset used and the approach taken to extract relevant data. Section 3 will discuss the results of our analysis, and answer the research questions listed above. Section 4 will contain a meta-discussion of the results and their greater meaning both for the Maven system and software development as a whole. Section 5 will contain a brief discussion of related work and how it relates to our work here. Finally, section 6 will cover threats to validity, possible avenues for future work, and a summarizing conclusion.

2 Methodology

Following the requirements of the MSR 2025 Challenge, we primarily used the provided dataset; this dataset has a focus on dependencies and the dependency ecosystem as a whole, to be interfaced with using the Goblin framework. Goblin itself is composed of a Neo4J dependency graph covering the Maven Central architecture, and includes a tool called Weaver to populate the dependency graph with more up-to-date data. We did not use the Weaver tool, simply using the provided dataset and interfacing with the graph using Neo4J.

Within the dependency graph, there are nodes for libraries, also referred to as Artifact nodes. Each Artifact presents itself as a general library; specific releases for that library in version numbers are separate nodes, known as Release nodes, which are connected to the base Artifact node. Additional nodes, called AddedValue nodes, are part of an augmented dataset that was used for this study. They contain additional information for Releases, such as Common Vulnerabilities and Exposure (CVE) information or popularity metrics.

Neo4J uses the Cypher querying language to retrieve database entries. Thus, the first step for all three research questions was to formulate appropriate queries that would give us the data we need to perform statistical analyses.

2.1 Research Question 1

To answer this research question, we decided to look at the first 2000 earliest releases in comparison to the 2000 latest releases. Our queries retrieved either the earliest or latest releases respectively, as well as their dependency counts. The queries themselves are as follows:

```
MATCH (r:Release)-[e:dependency]->(dep)
WHERE r.timestamp<1431653069000
RETURN r.timestamp, COUNT(dep),
r.id ORDER BY r.timestamp ASC
```

Listing 1: Retrieve 1000 Earliest Releases

```
MATCH (r:Release)-[e:dependency]->(dep)
WHERE r.timestamp>1700000000000
RETURN r.timestamp, COUNT(dep),
r.id ORDER BY r.timestamp DESC
```

Listing 2: Retrieve 1000 Latest Releases

Querying on Neo4J gave only the first 1000 entries, so we first collected the first 1000, and then skipped the first 1000 entries using the `SKIP 1000` command on a subsequent query to get 2000 entries in total. After retrieving these entries, we took both the mean and the median dependency count for each individual sample.

For all three research questions, we were also curious to see if there were any differences between more popular libraries in the Maven architecture and the general data retrieved normally. Thankfully, the AddedValue nodes in the augmented dataset have a metric for the popularity of a library. The `POPULARITY_1_YEAR` value is computed by measuring the number of dependants of a version of the library across a one year window, from the date the dependency graph was created. To retrieve such entries, we used the following query:

```
MATCH (pop:AddedValue)
WHERE pop.type="POPULARITY_1_YEAR"
RETURN pop.id, pop.value
ORDER BY toInteger(pop.value) DESC LIMIT 100
```

Listing 3: Retrieve 100 Most Popular Releases

We then looked at individual libraries retrieved from this query using a snowball sampling method to collect relevant data and then performed our statistical analysis on it. We used the following query to retrieve all the releases, dependency counts, and CVE information for these popular libraries:

```
MATCH (a:Artifact)-[e:relationship_AR]->(r:Release)
WHERE a.id=~"org.jetbrains.kotlin:kotlin-stdlib"
WITH a,r WHERE (r)-[f:dependency]->()
WITH count(f) AS depCount, a, r
MATCH (r)-[f:addedValues]->(g:AddedValue) WHERE g.type="CVE"
RETURN r.id, r.timestamp, depCount,
size(split(g.value,"cwe"))-1 AS CVECount
ORDER BY r.timestamp ASC
```

Listing 4: Retrieve All Releases and Relevant Metrics

Note that the `ai.id` parameter changed based depending on the library required. Some libraries, such as `org.jetbrains.kotlin:kotlin-stdlib` or `org.mockito:mockito-core`, did not have enough releases or dependencies to perform any form of statistically significant analysis on, so they were not considered.

2.2 Research Question 2

To answer our second research question, we needed to retrieve release data for libraries that had releases both very early and very late into Maven's lifespan. This would help illustrate the difference in release frequency across these two periods, and allow us to calculate if there has been any large change in mean or median in those counts.

We first retrieved all libraries and their associated releases that were released before March 13th, 2012, and then retrieved each respective library's releases after November 11th, 2023. These releases were all sorted based on their timestamps. We then took the two oldest releases and the two most recent releases for each library and performed a statistical analysis on the time between each respective release pair. The query to retrieve this data is as follows:

```
MATCH (a:Artifact)-[er:relationship_AR]->(r:Release)
WHERE r.timestamp>1700000000000 OR
r.timestamp<1331653069000
WITH a,r ORDER BY r.timestamp DESC
WITH a, collect(r) AS releases
WHERE releases[1].timestamp>1700000000000 AND
releases[size(releases)-2].timestamp<1331653069000
RETURN a.id, releases[0].timestamp,releases[1].timestamp,
releases[size(releases)-1].timestamp,
releases[size(releases)-2].timestamp
```

```
ORDER BY releases[0].timestamp DESC
```

Listing 5: Retrieve 1000 Latest Releases With Early Releases, Return Two Oldest and Two Newest

Similar to RQ1, we also looked at some of the most popular libraries using the same data collected in the manner described in the previous subsection. Here, we were looking to see if time between releases correlated with dependency counts; both libraries that had an overall decrease in dependency counts, and those that had an overall increase in dependency counts. Results will be expanded upon in the next section.

2.3 Research Question 3

We were unable to establish a baseline metric for CVE security vulnerability data in the AddedValue nodes, as the dependency graph's structure makes it infeasible to collect such data across all libraries. While we were able to assess individual libraries, establishing a baseline was beyond the scope of this work.

Given we were able to look at and assess individual libraries, we attempted to analyze this facet using the popular libraries as done in the last two research questions. This was still difficult, as libraries with CVE data are relatively uncommon in general; this extends to the popular libraries as well.

3 Results

In this section we will go over the results of our statistical analysis, looking at both the general trends within Maven, as well as the trends of some of its most popular libraries.

3.1 Research Question 1

The results from our analysis showed a great degree in variation in dependency counts across libraries over time. Plotted here are the dependency counts of the 2000 earliest releases, followed by the a plot of the dependency counts of the 2000 latest releases.

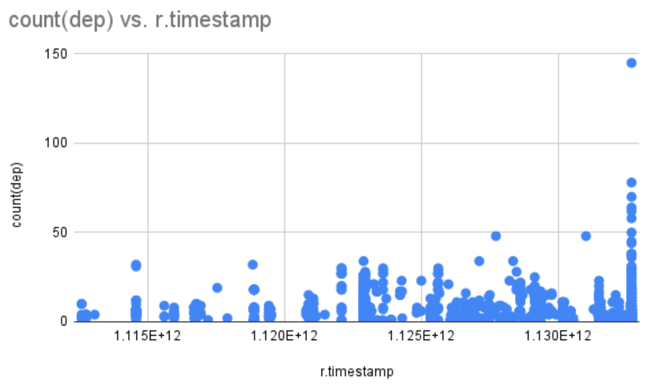

Figure 1: Dependency Counts 2000 Earliest Releases

Computing the mean value for dependency counts results in 7.11 average dependency counts in the earlier releases, compared to the average of 9.1265 for the later releases. Taking the median gives us 5 for the earlier releases, and 7 for the later releases. This

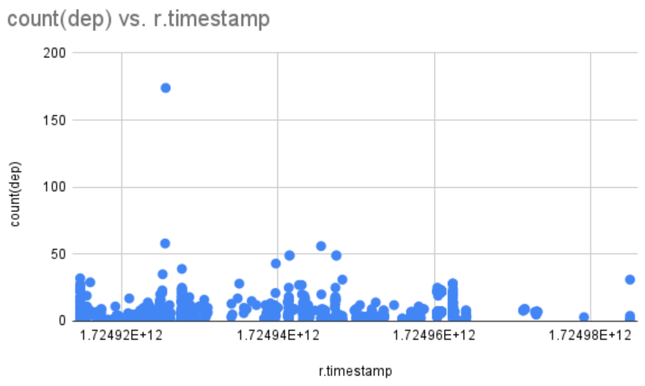

Figure 2: Dependency Counts 2000 Latest Releases

demonstrates that, generally speaking, Maven is suffering from a degree of dependency bloat.

However, looking at the more popular libraries gives more varied data. Table 1 below shows the values for dependency count from the first release, the last release, and also includes both the zenith and the nadir of the counts as well. `org.apache.logging.log4j:log4j-core` suffers some dependency bloat, to a high of 57, before quickly cutting down to a value of 24. Notably, there is a large drop in dependencies at 2.20.0.

`org.junit.jupiter:junit-jupiter-engine` starts from a high of 10, before quickly trimming down to 3 dependencies, with some fluctuation. There does not seem to be a correlation between dependency reduction and major version count for this library. `org.junit.vintage:junit-vintage-engine` is a very similar library, and thus has very similar results, with slightly differing highest and lowest values. Again, not much correlation between version change and dependency count.

On the other hand, `ch.qos.logback:logback-classic` instead suffered from dependency bloat, reaching a high of 26 before abruptly cutting down to a steady count of 20. It's worth noting that pre-1.0 release versions began at 3 dependencies, bloating to 24 on full release, followed by a steady decline. `com.fasterxml.jackson.core:jackson-databind` and `commons-io:commons-io` also suffered dependency bloat; the latter library had a large timescale between going from 1 dependency to 5, suggesting a possible refactoring of the library.

3.2 Research Question 2

The results of our analysis of release speeds show that time between releases has indeed decreased overall. The timestamp data is in milliseconds from the Unix baseline, so it has been converted to days for convenience. In the latest releases, we find that the mean time between the two latest releases of a library is 3687004220 (42.67 days), whereas the median is 1550125500 (17.94 days); this gives us a variance of approximately 2.70521E+19.

For the earliest releases, the mean time between two releases was 5433446102 (62.87 days), with a median release timeframe of 3110888000 (36.006 days). This gives us a difference between the mean time for the latest and the mean time for the earliest of

Library Name	Start	End	Start-End %	Low	High
`org.apache.logging.log4j:log4j-core`	13	24	185%	12	57
`org.junit.jupiter:junit-jupiter-engine`	10	3	-67%	3	10
`org.junit.vintage:junit-vintage-engine`	12	3	-75%	2	12
`ch.qos.logback:logback-classic`	3	20	667%	3	26
`com.fasterxml.jackson.core:jackson-databind`	8	12	50%	6	12
`commons-io:commons-io`	1	10	1000%	1	10

Table 1: Dependency Changes for Selected Libraries

1746441882 (20.213 days), giving us a factor of 0.6785756499. Doing the same for the median gives us a difference of 1560762500 (18.064 days), and a factor of 0.4982903595.

Such results are a clear indication that releases have increased in frequency over Maven's lifetime. The variance for both results is quite high, making the median a better metric for measurement. Even so, both metrics demonstrate that time between versions has decreased by a significant amount; a factor of nearly half.

Looking at the popular libraries again, notably those with reductions in dependencies, we also wanted to view if there were differences in trends between major releases and minor releases; major releases being those with a new initial number in the version count, and minor releases being those that increment a subsequent number, pre-releases, milestones, and release candidates. `org.apache.logging.log4j:log4j-core` trends upwards in time between releases, suggesting a slower release schedule. In terms of overall metrics, the mean time between releases was 6002281672 (69.47 days), with the mean time for a major patch being 9298893667 (107.63 days) and the mean time for a minor patch being 3863938757 (44.72 days). This leads to a factor of approximately 2.4, which explains the zig-zag nature of the chart as seen below in Figure 3.

`org.junit.jupiter:junit-jupiter-engine` has a similar looking graph; an upward trend in time between releases, with a mean across all releases of 3706448855 (42.9 days). The mean time for a major patch is 1334937500 (15.45 days), while the mean time for a minor patch (including milestone and release candidates) is 4205714404 (48.68 days). Again, a factor of approximately 0.31 reflects the zigzag nature of the chart. Interestingly, `org.junit.vintage:junit-vintage-engine` has an identical release schedule, so the same considerations apply.

On the other hand, `ch.qos.logback:logback-classic` reflects the hypothesis to a high degree, with big drops in between releases correlating with drops in dependency counts. However, it also correlates with major release version, thus leading to an unclear conclusion. The mean time between releases was 4202877133 (48.61 days), with the mean time for a major patch being 3986234250 (46.18 days) and the mean time for a minor patch being 4224013024 (48.84 days). This indicates a downwards trend in time between releases.

The libraries with dependency bloat did not demonstrate any interesting correlation, and thus have been abrogated. Even among the library expanded upon above, there doesn't seem to be a measurable trend across the popular releases.

3.3 Research Question 3

As explained earlier, we found it infeasible within the time given to work with the CVE data across all the libraries within Maven. As such, we were unfortunately unable to answer this question ourselves. The popular library we looked at were also largely missing CVE data. Regardless, we were able to look at some of the libraries and establish trends.

`org.apache.logging.log4j:log4j-core` had an increase in time between releases, but had an increase in CVE reduction frequency. `ch.qos.logback:logback-classic` has an initial increase in time between releases, before speeding up and greatly reducing the time between releases. CVE count was observed to change quickly, suggesting that, while quick patches may introduce security vulnerabilities, they can also be responded to quickly.

4 Discussion

RQ1 was an exercise in seeing if general best practices regarding dependency counts were being followed in the Maven ecosystem. We hoped to see that dependency counts would trend downwards, as to maintian code quality and mitigate security vulnerabilities. In practice, and in retrospect, seeing dependency bloat increase should have been expected.

There is of course the primary argument of software systems getting more complex from the early 2010s to the mid 2020s of today. There are far more considerations and expectations for systems today, which logically lends itself well to the trends seen. Furthermore, it's likely many of these libraries accumulated technical debt in the form of dependency bloat; even some of the more popular libraries struggle to keep dependency count from increasing over time. However, given the average of minimal changes in dependency counts amongst popular libraries, there is an illustrated effort to combat dependency bloat.

However, RQ2 did indeed reflect the industry trend of release frequency increasing overall; there is much empirical evidence of this trend both within the dataset, and across industry. More interesting is the results from the popular libraries. One would intuit that major releases would take longer to release than minor ones, as the expectation is that minor releases can be done more quickly to patch up bugs or flagged vulnerabilities.

This is indeed the case with some libraries, such as `org.apache.logging.log4j:log4j-core`; this is not the case with other libraries such as `ch.qos.logback:logback-classic` or the two jupiter-engine libraries. Closer inspection reveals that these libraries have many releases that seem to "bridge" the time between major releases. This includes steady maintenance patches for older

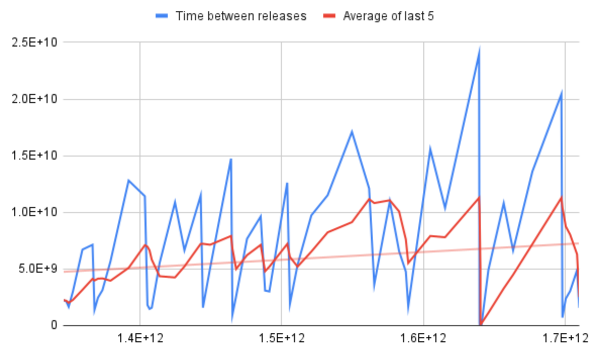

Figure 3: org.apache.logging.log4j:log4j-core

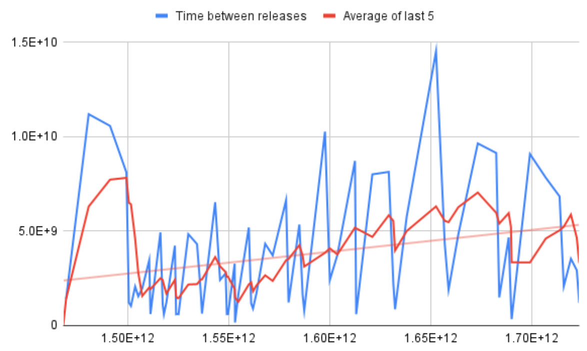

Figure 4: org.junit.jupiter:junit-jupiter-engine

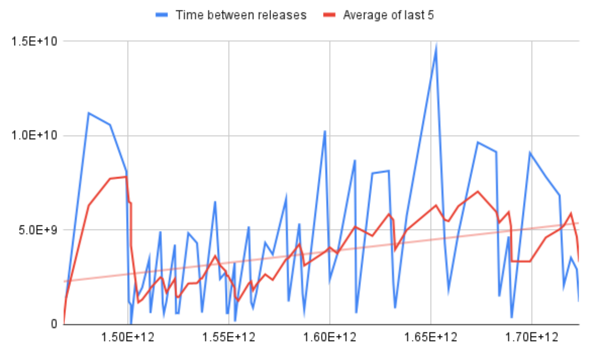

Figure 5: org.junit.vintage:junit-vintage-engine

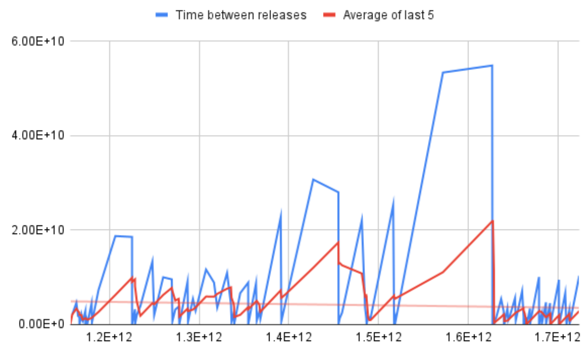

Figure 6: ch.qos.logback:logback-classic

Figure 7: Four Graphs Plotting Time Between Releases and Average of the Time Between the Last 5 Releases

releases, pre-release versions, milestones, and release candidates. Further discussion will occur in Threats to Validity section.

As previously mentioned, RQ3 was infeasible for us to answer within the time given, given the structure of the data within the AddedValue nodes. To further elaborate, the CVE data is stored as a JSON string within the AddedValue node as a parameter. Cypher cannot read JSONs, so extracting severity and id values would have required additional extraction techniques given we found that Cypher cannot parse these values. This made it very difficult to reason about trends in CVE data using these metrics without first extracting all possibly relevant nodes, and then extracting the data within them in a different way. Further possible ways this could have been done will be discussed in the Future Work section.

5 Related Work

Given our dataset is from the MSR 2025 Mining Challenge, there are some submissions to the challenge that are related to the work we completed here. Rabbi et al. [2] take a closer look at the security vulnerability information in the AddedValue nodes, assessing how quickly the vulnerabilities are fixed in the Maven ecosystem relative to their severity, the library's popularity, and release frequency. Similarly, Shafin et al. [3] explore how release speeds correlate with software security and release frequency, and did find that faster release cycles were linked to fewer CVE counts in dependency chains. Suwanachote et al. [5] focus instead on Java projects, looking to see how many projects have packages that go unused, or if any dependencies introduced into a project never see use.

Outside of the MSR domain, there has been much work done on tracking both dependencies and release speeds. Wang et al. [6] take a closer look at streamlining bloated dependency structures in Java projects. They introduce a technique they call "slimming"" to reliably remove dependencies from projects that are causing needless bloat. Khomh et al. [1] explore the shift from more traditional long release cycles that would take multiple months, to the now more common shorter release cycles. Looking closer at Mozilla Firefox, they compare crash rates, median uptime, and the proportion of pre- and post-release bugs between releases in the traditional timeframe and the contemporary faster release cycle. They also assess if there have been changes to the development process by analyzing the source code across these periods.

Dependency analysis, the impact of release cycle timeframes, and security vulnerability analysis are all hot-button topics in the field, and will only see more analysis as we both seek to achieve generality in results, while also creating new products and methodologies that shake up the field.

6 Summarizing Thoughts

Before we summarize the work we have done, we will first take a look at the possible threats to validity that arose while we worked towards our results.

6.1 Threats to Validity

One limitation of this study is the ordering of releases by timestamp, which makes the interpretation of time between releases only partially reliable. In many cases, multiple versions of a library receive updates simultaneously. For instance, in the case of `log4j-core`, versions 2.3.2 and 2.12.4 were released after 2.17.1, indicating that different resources were allocated to separate releases. This complicates the meaning of time between releases, as updates may not be directly related. A more refined approach would involve distinguishing between releases that received updates after subsequent versions, under the assumption that they utilize separate resources. However, this approach falls outside the scope of this study and would significantly reduce the available dataset while introducing further complexities.

Another challenge arises from pre-release versions, including those marked as alpha, beta, Mx (milestone x), or RCx (release candidate x). These versions act as transitional stages between full releases and can distort data, particularly when analyzing time between releases. For example, for `junit-jupiter-engine`, the time span from 5.3.2 to 5.4.0 appears to be 6.4E+09, but when accounting for an intermediate milestone and two release candidates, the immediate preceding release's timestamp places this time span at only 6.07E+08; a difference by a factor of 10. To obtain a clearer picture of release intervals, it may be preferable to exclude all pre-release versions, and possibly even minor version updates.

Additionally, the structure of the dependency graph presents a challenge. There is no way to confirm whether a particular release or library is actively used within the Maven system, as the core dataset only provides timestamps. This makes it difficult to empirically assess the impact of certain releases. When a minor version update for an older major release enters the system, its continued relevance remains unclear, complicating any conclusions drawn from dependency trends. Even with the augmented dataset, the AddedValue nodes only contain popularity metrics for the previous year, from when the graph was generated. Generating a new graph with popularity metrics stretching out to the early days of Maven was beyond the scope of this study, and it is unknown if such data is even available that far out.

Finally, the sheer scale of the dependency graph introduces a potential for human error in data interpretation. With 15,117,217 nodes and approximately ten times as many edges, extracting meaningful patterns and generalizable insights is inherently difficult. The complexity of the graph structure makes it challenging to ensure complete accuracy in both data collection and analysis.

6.2 Future Work

In terms of future work, the most clear path would be to address the issues mentioned in the threats to validity above. Iterating on the techniques used in this study while addressing the issues in the data as we extracted it would hopefully lead to more clear statistical results. This would help us make conclusions with more confidence, as while we do see correlation with many of our results, the veracity of such correlation may be impacted by the somewhat messy data. Another approach may be to sort entries based on version number, as opposed to timestamps; as we saw, older version updates are often released after a newer version has been rolled out.

While we initially wanted to do more with the CVE data, we found it difficult to work with and parse within the time provided. Given more time and resources, it would be interesting to see if there is any correlation between the previously analyzed metrics, dependency count and release speeds, against CVE metrics; these include the number of CVEs, the response time to clearing CVEs, and how these metrics change with CVE severity. We see from the related work that CVE severity does correlate with response time, so we would both attempt to repeat that result and looking more granularly at other metrics. Further investigation into driver technologies for Cypher and Neo4J would also help guide efforts on this front.

Finally, we did see that popular libraries did have some differences in comparison to the general data we extracted; however, we did not find enough to make any convincing conclusions. Performing a more rigorous analysis between more and less popular libraries and comparing metrics such as release frequency or CVE response time may provide interesting results.

7 Conclusion

Our study sought to understand the evolution of dependency structures within the Maven ecosystem and its broader implications for software architecture. The findings challenge the assumption that dependency counts naturally decline over time, instead highlighting a general trend of dependency growth, with certain libraries demonstrating later-stage optimization. Furthermore, while release frequencies have accelerated, we found no strong correlation between dependency count reduction and faster release cycles. Security vulnerability assessment was constrained by dataset limitations, leaving open the question of how dependency trends impact vulnerability management at scale.

These results underscore the complexity of dependency management in modern software ecosystems. While minimizing dependencies remains an ideal goal for maintainability and security, real-world constraints—such as feature expansion and external library reliance—complicate this effort. Our analysis of popular libraries suggests that dependency bloat is often followed by strategic reductions, reflecting a balancing act between functionality and architectural streamlining. Future work should explore finer-grained metrics for dependency utility, more comprehensive security vulnerability assessments, and broader implications for software maintenance practices. By deepening our understanding of these trends, developers and researchers can work towards more sustainable and secure software ecosystems.

References

[1] Tejinder Dhaliwal Ying Zou Foutse Khomh, Bram Adams. 2014. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering* 19, 6 (2014), 1191–1234. doi:10.1007/s10664-014-9308-x

[2] Rajshakhar Paul Minhaz F. Zibran Md Fazle Rabbi, Arifa Islam Champa. 2025. Chasing the Clock: How Fast Are Vulnerabilities Fixed in the Maven Ecosystem?. In *Proceedings of the MSR 2025 Mining Challenge*. https://2025.msrconf.org/details/msr-2025-mining-challenge/16/Chasing-the-Clock-How-Fast-Are-Vulnerabilities-Fixed-in-the-Maven-Ecosystem- Accessed: 2025-03-30.

[3] S. M. Mahedy Hasan Minhaz F. Zibran Md Shafiullah Shafin, Md Fazle Rabbi. 2025. Faster Releases, Fewer Risks: A Study on Maven Artifact Vulnerabilities and Lifecycle. In *Proceedings of the MSR 2025 Mining Challenge*. https://2025.msrconf.org/details/msr-2025-mining-challenge/6/Faster-Releases-Fewer-Risks-A-Study-on-Maven-Artifact-Vulnerabilities-and-Lifecycle Accessed: 2025-03-30.

[4] MSR 2025 Mining Challenge. 2025. MSR 2025 Mining Challenge. https://2025.msrconf.org/track/msr-2025-mining-challenge?#event-overview Accessed: 2025-03-30.

[5] Yutaro Kashiwa Bin Lin Hajimu Iida Nabhan Suwanachote, Yagut Shakizada. 2025. On the Evolution of Unused Dependencies in Java Project Releases: An Empirical Study. In *Proceedings of the MSR 2025 Mining Challenge*. https://2025.msrconf.org/details/msr-2025-mining-challenge/22/On-the-Evolution-of-Unused-Dependencies-in-Java-Project-Releases-An-Empirical-Study Accessed: 2025-03-30.

[6] Hai Yu Zhiliang Zhu Ying Wang, Shing-Chi Cheung. 2025. Streamlining Software: Bloated Dependencies. *ResearchGate* (2025). https://www.researchgate.net/publication/389326457_Streamlining_Software_Bloated_Dependencies Accessed: 2025-03-30.