Thesis

Identifying, documenting, and investigating breaking changes in software ecosystems

Christoforos Papandreou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2025

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Identifying, documenting, and investigating breaking changes in software ecosystems

Christoforos Papandreou

Supervisor

Dr. Eleni Constantinou

The Thesis was submitted in partial fulfillment of the requirements for obtaining a degree in the Department of Computer Science of the University of Cyprus

May 2025

Acknowledgements

I would like to express my great gratitude towards my supervisor, Dr. Eleni Constantinou, for giving me the chance to work on this topic and for guiding me through all the steps needed for the making of this very thesis.

Additionally, I want to thank the people that are close to me and have supported me emotionally when they felt that they needed to. I will always be grateful to you.

Abstract

In the field of software evolution, dependency library updates may introduce changes which interfere with client-dependency compatibility. Therefore, client developers could possibly delay updating their dependency constraints and thus cause technical lag. Technical lag represents a value that describes the outdatedness of a client's dependency constraints based on the time of its release. This thesis explores the interconnection between three key variables of software evolution via setting these three research questions: (1) Does release frequency affect the appearance of breaking changes? (2) Do breaking changes affect technical lag? (3) Does release frequency influence technical lag?

To investigate the three RQs, this thesis builds upon the dataset of [4] and extends it with more recent Maven Central Repository data, such as release dates and newer release versions. Any unclear or invalid entries are filtered out of the expanded dataset with the use of multiple scripts. Values for breaking changes between library updates were already pre-calculated in [4], while release frequency, technical lag and version lag were all calculated within this thesis.

Using the Spearman's Rank Correlation test and the newly defined datasets, the three RQs were explored, and they revealed some interesting results. They revealed that RQ1 and RQ2 had a statistically significant correlation, albeit it being weaker than expected, while RQ3 showed no correlation between its two variables.

The findings indicate that despite the fact that an increase in library release frequency leads to less added breaking changes in the code, several other factors matter for the number of breaking changes in a library update. A similar conclusion can also be given about the fact that updates which include more breaking changes are more likely to lead their clients to higher levels of technical lag, but they are not the sole factor for this. As for RQ3, there does not seem to be a direct connection between dependency release frequency and client technical lag, which can also be an indication of multiple external factors, such as developer behavior, affecting the relationship of the three key variables.

In conclusion, this thesis contributes to understanding that the connection between the three key variables of software evolution is not as simple as it may seem and also proposes future directions for further investigation in this field.

Table Of Contents

Chapter 1

Introduction

Technological developments have made it more urgent for software to change and grow in order to satisfy user and market expectations. For software systems to be maintained, improved, secure as well as to be effective and functioning, these ongoing updates are essential. [4] Many commonly used software libraries have undergone several revisions and upgrades in recent years, all aimed at adding new features, addressing bugs, enhancing security, or improving speed. The aforementioned changes, however, despite their importance, often pose serious risks, particularly for systems and projects that rely on them. [1, 4, 5, 8]

Keeping a project up-to-date with its dependencies should in theory be of vital importance. When dependencies release a new version, developers must make a choice. Either to use it or skip it. While an update often introduces new or improved features, chances are that it will not be compatible with the project, due to the appearance of breaking changes - changes that can cause build failures and runtime errors. [4, 5] As a result, some developers may skip new dependency releases or choose to temporarily ignore them. This leads to what is known as technical lag, a situation where dependency outdatedness is present in a project, often depriving it from safety and effectiveness. [2, 6, 7, 9]

Technical lag, in definition, refers to the amount of time by which a client lags behind the latest available version of a dependency at the moment of its own release. [7]. It reflects on the challenge of balancing stability and the need to stay up-to-date. Over time, projects with high technical lag may face problems with their compatibility with more recent software or miss out on performance and security. [2, 3, 9]

The fear of breaking changes is one of - if not the most - important factors of technical lag. [4, 6] Developers are often aware of future compatibility problems, however they choose to ignore them whenever there are no apparent benefits of using a dependency update. This interplay between release frequency, breaking changes, and technical lag is at the heart of this thesis.

All three of these factors have been investigated previously. However, the relationship between all three remains unclear, and this thesis aims at uncovering the potential connections and correlations between these factors, guided by three main research questions: (1) Does the release frequency relate to the appearance of breaking changes? (2) Do breaking changes relate to technical lag? (3) How does the frequency of software releases affect technical lag?

The dataset used in this thesis comes from the Maven Repository, a widely-used platform for managing dependencies in Java-based projects [4]. Maven plays a key role in modern software development by helping developers manage, share, and distribute software libraries. With its extensive collection of library versions, detailed metadata, and a clear timeline of past releases, Maven provides a valuable representation of how software evolves over time. [4, 5] This makes it an ideal resource for studying real-world patterns in how dependencies change and how projects respond to those changes.

By exploring these three questions, using statistical tests such as the Spearman's Rank Correlation test, this thesis aims to provide a clearer understanding of the connection between the three variables. Furthermore, it should help developers make smarter decisions about when to update and eventually lead to more polished software ecosystems.

Chapter 2

Related Work

In recent years, several related studies have been published from which this thesis draws inspiration and attempts to build upon. Many researchers have explored the concepts of software evolution, technical lag, breaking changes and release frequency. However not many have touched on the interconnection of all the previously mentioned metrics. This thesis aims to investigate and find a clearer connection between all three.

2.1 Studies on Technical Lag

Zerouali et al. [9] explored the relationship between technical lag and breaking changes, suggesting that technical lag results from developers postponing updates due to the fear of breaking changes in newer versions of dependencies. However, by doing so, they unintentionally put their libraries at risk, often making them vulnerable and insecure. This thesis incorporates version lag into the analysis, providing a more complete interpretation of technical lag. When version lag indicates that the skipped updates are insignificant, despite a high technical lag value, it allows developers the benefit of the doubt, acknowledging that their hesitation to update may not always be damaging to a project.

Decan et al. [2] completed a research on thousands of npm packages, questioning the chronic trends of technical lag appearance within them, and finding that technical lag increases over time. However, they also noted that when SemVer principles are met, the values for technical lag tend to decrease.

Similarly, Stringer et al. [7] investigated the technical lag of multiple packages and noticed that developers tend to prefer a more stable release rather than the uncertainty of frequent updates.

While all the aforementioned studies investigate technical lag, this thesis aims to extend on their work by correlating technical lag with breaking changes and release frequency altogether using MCR packages and finding out how each of the attributes affect one another.

2.2 Studies on Breaking Changes

Raemaekers et al. [5], conducted a research on breaking changes of the MCR and suggested that they do not only appear in major releases, but also in minor and in patch ones. This is due to many developers not following the SemVer principles, making package maintenance complicated.

Similarly, Xavier et al. [8], discussed issues related to the incorrect use of SemVer principles and later on Ochoa et al. [4] built upon these studies. They explored and calculated breaking changes in MCR using the Maracas tool. They suggested that many projects do not follow the SemVer principles, resulting in breaking changes in versions that they should not appear. While Ochoa et al. [4] identifies and records the breaking changes, this thesis explores their relationship with technical lag and release frequency, thus extending their work.

Additionally, Robbes et al. [6] through their study, they found that breaking changes lead to technical lag, more often than not, and that there is a direct connection between the two.

What this thesis contributes to the previously mentioned studies on breaking changes is the chance to not only investigate the connection between breaking changes and technical lag, but to also incorporate information about release frequency as well in the equation.

2.3 Studies on Release Engineering

Xavier et al. [8] examined the correlation between release frequency and breaking changes, and suggested that more frequent releases typically lead to the appearance of more breaking changes.

This thesis contributes and extends Xavier et al.’s work by also correlating technical lag with both the release frequency and the breaking changes, thus showcasing their effects on the developers’ decision to not update as frequently as they should.

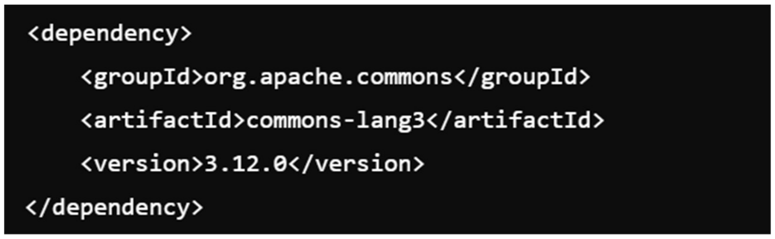
Chapter 3

Background

3.1 Maven Central Repository (MCR)

Maven Central Repository is a public repository which is widely used by Java developers. It acts as a centralized hub where Java projects can be uploaded and downloaded. MCR simplifies the downloading process of projects by requiring a specific file to exist within them that declares all the dependencies of the project. This file is usually named "pom.xml" and it follows the structure that can be identified in Figure 1.

Libraries in MCR are organized using a hierarchy/directory based system that separates them firstly by group, then by artifact and lastly by version number, making the searching process easier and quicker for the user.



```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
</dependency>
```

Figure 1

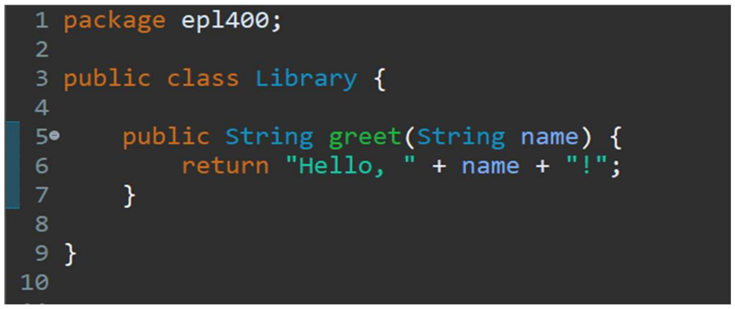
Dependency section of the pom file of a library version

Apart from this structure, it is also possible that the dependency version can be set through the "parent" section of the pom file, which allows child projects to automatically use dependency versions defined in their parent project's pom file.

3.2 Breaking changes

Breaking changes are modifications that are made to a software library, framework, or API in comparison to its previous versions. These modifications are the cause of failure of systems that depend on the libraries with the breaking changes. They disrupt backward compatibility and are the explanation for a potentially unexpected behavior from the libraries that depend on them.

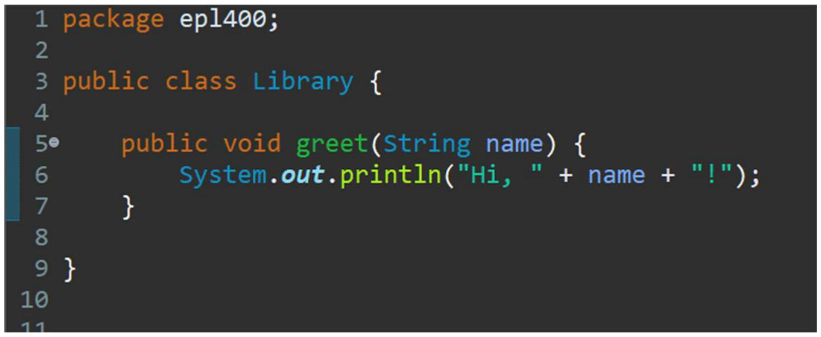
Removing or renaming a method, class, or field that was previously exposed and used by clients, or changes to method signatures, such as altering parameters, return types, or visibility levels are all major types of breaking changes. Other types include instances where the functionality or output of a method is altered in terms of prior expectations, or through dependency changes, like replacing or removing components in which the client systems rely on.



```java
package epl400;

public class Library {

    public String greet(String name) {
        return "Hello, " + name + "!";
    }

}

```

Figure 2

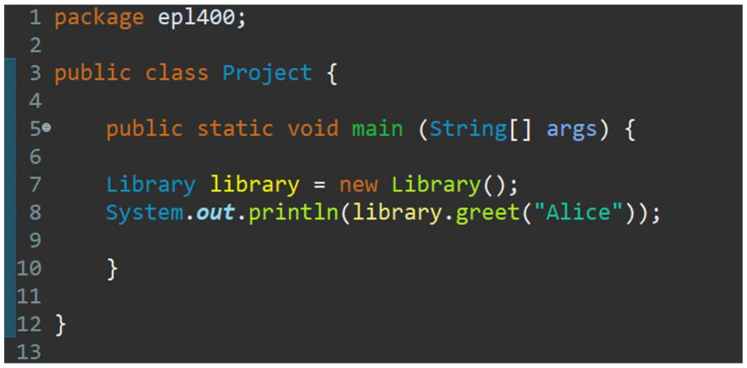
Library version using a string method return type for method “greet”



```java
package epl400;

public class Library {

    public void greet(String name) {
        System.out.println("Hi, " + name + "!");
    }

}
```

Figure 3

Library version using a void method return type for method “greet”



```java
package epl400;

public class Project {

    public static void main (String[] args) {

    Library library = new Library();
    System.out.println(library.greet("Alice"));

    }

}
```

Figure 4

Project depending on the “Library” class

Looking at Figures 2, 3 and 4, an example of a breaking change can be identified if we assume that Figures 2 and 3 are two different versions of the same library. In Figure 2, the “greet” method is constructed with a return type of “string” and upon its calling by the “main” method in the “Project” library in Figure 4, no errors would appear. The output would appear as expected. However, if the “main” method was to call the “greet” method in Figure 3, which is of type “void”, a compilation error would occur, as nothing is returned back to the “main” method. This type of breaking change is known as a “method return type change”.

Furthermore, with the possibility of updated libraries causing failures or unexpected behavior to a client system, a plethora of developers choose to either not update their dependencies at all, or postpone the updating until further notice. Henceforth, technical lag appears in such cases.

3.3 Technical (time) lag

According to Zerouali et al [9], “technical lag captures the delay between versions of software deployed in production and more recent compatible versions available upstream”.

The technical lag of a client system with respect to a dependency is defined as a temporal measure that estimates how outdated a dependency of the client is being used by the client at the time of its release. It can be categorized into two cases:

Mathematically, for a client C dependent on a version V of a dependency:

$$\text{Technical Lag} = \begin{cases} 0 & \text{if } V \text{ is the latest version released before or on } X, \\ \text{Date}_{\text{Next Version}} - X & \text{if } V \text{ is not the latest version released before or on } X. \end{cases}$$

Following the calculation of the technical lag that is caused to the client by its every dependency, the overall technical lag of a single client is agreed to be the largest one that emerges out of all its dependencies.

Zerouali et al. [9] proposes a formal framework for measuring technical lag. The term “time lag” is used to define the temporal lag described above. Henceforth, in the thesis the term “technical lag” will refer to the temporal definition of time lag, unless otherwise stated.

3.4 Version Lag

Version lag is a measurement used as an alternative way of measuring a library's lag. It consists of three different lag categories, namely the major one, the minor and the patch. A single unit is added to one of the categories for each of such later versions of a dependency that were ignored by a client.

In cases where no technical lag appears for a client - dependency pair, the values for each version lag category are also equal to zero.

However, if technical lag exists, for each major version ignored by the client, the major type of version lag is incremented. That is also the case for every minor version and the minor type of version lag, as well as every patch version and the patch type of version lag.

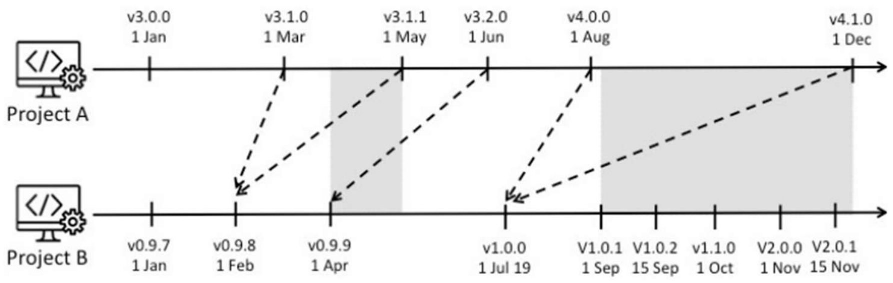


Figure 5

Example of a dependency. Originally from [7]

As an example, in Figure 5, which appears in [7], the client version 4.1.0 depends on version 1.0.9 of Project B. This precisely means that the major version lag of version 4.1.0 of the client would equal to 1, as the client depends on version 1.0.0 of Project B and the major update 2.0.0 of the dependency was ignored. The minor version lag would also equal 1, since the 1.1.0 update of project B was skipped. Finally, the patch version lag of this version of project A would equal 3, as the versions 1.0.1, 1.0.2 and 2.0.1 of project B were overlooked.

Two libraries may have the same technical lag, but that does not mean that the updates of the dependencies which they ignored are of the same importance.

Therefore, version lag is a complementary metric of technical lag that provides a clearer assessment of the actual impact of an outdated dependency by distinguishing whether the skipped updates are bug fixes, added features or major breaking changes.

3.5 Spearman's Rank Correlation Test

The Spearman Rank Correlation Test is a statistical method used to determine the strength and direction of a relationship between two variables which does not assume any specific distribution of the data. It focuses on the ranks of the values rather than on the raw numbers.

It works by ranking both variables from lowest to highest. For each pair of data points, the difference between their ranks is calculated. The test then computes a correlation based on these rank differences. A high positive correlation (close to +1) indicates that as one variable increases, the other tends to increase as well. A high negative correlation (close to -1) suggests that as one variable increases, the other tends to decrease. A correlation near 0 means there is little to no relationship between the variables.

In terms of statistical hypothesis testing, the null hypothesis assumes that there is no correlation between the two variables. Ultimately what will tell whether the coefficient is significant is the p-value. A p-value lower than a chosen significance level (usually 0.05) indicates that the coefficient is statistically significant, supporting the idea that there is a meaningful relationship between the variables, which did not happen by chance. However, if the p-value is greater than 0.05, the null hypothesis is not rejected, indicating that the observed correlation could be due to random chance.

3.6 Semantic Versioning (SemVer)

Semver versioning is used as a way of identifying the different versions of a library. It is composed of three integer values separated by dots. The format that is used is “MAJOR.MINOR.PATCH”. The major version integer is incremented when changes that could break existing functionality appear in the new version. The minor version integer is typically incremented when new features are added to the library that do not break functionality and are backwards compatible. Finally, the patch integer is incremented in the case where the libraries are updated for the purpose of bug fixes. A good example of the correct use of semantic versioning would be version 1.0.4 of any library evolving to 1.0.5 or even 1.1.0, if no breaking changes were added in the code. In the latter case, the new version should be 2.0.0.

However, not all libraries in Maven use semantic versioning correctly, and this introduces instances with obscure versioning, such as a really low major and minor value alongside a really high patch value in libraries where breaking changes have happened multiple times.

Chapter 4

Methodology

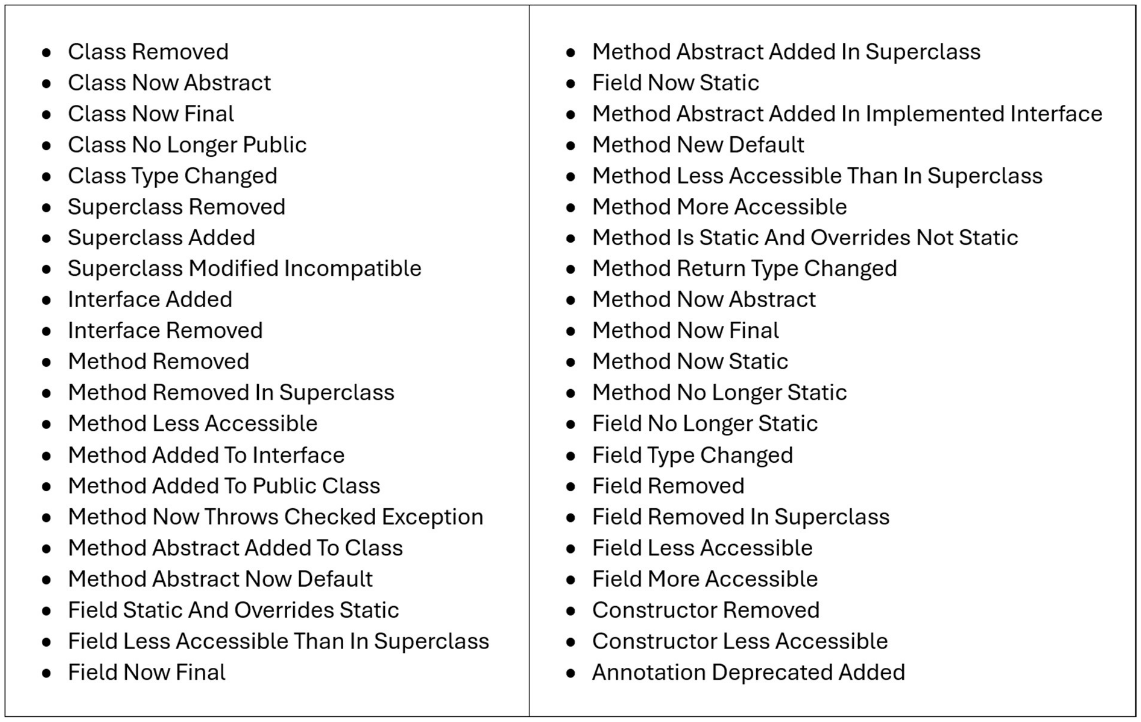
4.1 Initial Data filtering

This thesis extends the work done by Lina Ochoa et al. in [4], and therefore utilizes the data in the replication package. The dataset consists of libraries found in the Maven Central Repository before 2018 and includes data regarding the number of breaking changes throughout multiple MCR packages as well as their dependencies and their versioning numbers.

It must be mentioned that the breaking changes in the dataset have been identified and counted by the Maracas (Metric-based Analyzer for Refactoring and Change Assessment in Software) tool, which is a tool that provides detailed metrics for each breaking change type by comparing different versions of a library's API. Additional information about the tool can be found in [4], where the use of it is explained in detail by Lina Ochoa et al. The types of breaking changes that the Maracas tool can detect are listed in Figure 6.

Upon reviewing a file containing information about different versions of libraries, it was noticed that some versions were missing from the dataset. These versions had been released before 2018 and thus, it was decided that with the use of a script the missing versions could be added. Moreover, since an expansion was made to the dataset, it was decided that the most recent versions would be included as well in it. Later on, non SemVer versions were removed from the dataset, in order to focus on the SemVer ones. Additional scripts were also used to add useful information (release dates) to the dataset and others were used to filter out invalid records, where multiple versions of the same library claimed to have been released on the same date. The thought process and the correct running order of the scripts are included in the README file of the GitHub repository of this thesis, which can be found in Appendix A.

The information about the dependencies of each library in the dataset is a key component that is used for the calculation of the technical lag and version lag, which is needed in order to get the answer to the third research question of this thesis. The libraries that were missing and were added via script are also important for the calculation of a more accurate release frequency of the libraries.



- Class Removed
- Class Now Abstract
- Class Now Final
- Class No Longer Public
- Class Type Changed
- Superclass Removed
- Superclass Added
- Superclass Modified Incompatible
- Interface Added
- Interface Removed
- Method Removed
- Method Removed In Superclass
- Method Less Accessible
- Method Added To Interface
- Method Added To Public Class
- Method Now Throws Checked Exception
- Method Abstract Added To Class
- Method Abstract Now Default
- Field Static And Overrides Static
- Field Less Accessible Than In Superclass
- Field Now Final
- Method Abstract Added In Superclass
- Field Now Static
- Method Abstract Added In Implemented Interface
- Method New Default
- Method Less Accessible Than In Superclass
- Method More Accessible
- Method Is Static And Overrides Not Static
- Method Return Type Changed
- Method Now Abstract
- Method Now Final
- Method Now Static
- Method No Longer Static
- Field No Longer Static
- Field Type Changed
- Field Removed
- Field Removed In Superclass
- Field Less Accessible
- Field More Accessible
- Constructor Removed
- Constructor Less Accessible
- Annotation Deprecated Added

Figure 6

Types of breaking changes

4.2 Answering RQ1: Does the release frequency relate to the appearance of breaking changes?

To determine whether the release frequency of library versions relates to the appearance of breaking changes, first, the release frequency was defined accordingly, to match the pre-calculated breaking changes of a single library update from [4]. In this research question, the release frequency equals the amount of versions released from the very first release of a library up to the more recent release of a pair of consecutive versions, divided by the time difference between the two, in months. Using datasets found in [4] and the datasets that were expanded as mentioned in Section 4.1, the release frequency was calculated and the breaking changes between pairs of consecutive versions of libraries were retrieved.

Upon constructing the dataset, it was noticed that certain pre-calculated values of breaking changes were invalid due to compatibility issues which occurred in [4]. As a result, a decision was made to filter out all libraries which at any point of their evolution we had failed to retrieve their breaking changes.

Therefore, after this procedure, the dataset was finalized. To explore and analyze the relationship, this thesis has used the Spearman's Rank Correlation Test in order to measure the strength and the significance of the compared values.

4.3 Answering RQ2: Do breaking changes relate to technical lag?

To answer the second research question, the breaking changes values were retrieved from RQ1 and the technical lag values, as well as the version lag values were retrieved from RQ3. In cases of multiple dependencies per client, each dependency's contribution to technical lag was treated as a separate data point to isolate the direct impact of individual dependency changes.

Thus, the dataset was complete, in order to assist in determining the effect that the breaking changes that a library update introduced will have on the amount of technical lag of a client that depends on it.

We then applied Spearman's Rank Correlation Test to evaluate the strength and direction of the relationship between the number of breaking changes and the amount of technical lag. This type of analysis was chosen because it is reliable when investigating non-linear relationships which, in this case, are likely to be monotonic.

Additionally, a secondary analysis was performed to investigate how breaking changes relate to each type of version lag (major, minor, patch) to help contextualize the version-skipping behavior of clients and its potential impact on the severity of updates.

4.4 Answering RQ3: How does the frequency of software releases affect technical lag?

Using datasets that were available in [4], thousands of client – dependency pairs were retrieved. Using a script that can be found in this thesis' GitHub repository, the technical lag was calculated as described in Section 3.3, and a slightly different approach was used for the calculation of release frequency. In the case of RQ1, all library releases had to be accounted, but in the case of RQ3, the relevant libraries were the ones released after the release of their paired client. After the calculation of the newly defined release frequency, it was discovered that some libraries claimed to have been released after a client that uses them as dependencies, which is impossible. So, cases like these were assumed to be invalid, and were therefore removed from the dataset. In addition to the technical lag, the version lag of the client - dependency pairs is calculated in order to complement the technical lag measurements. Cases where major updates were missed are considered more significant than others where only minor or patch updates were missed. This is because major updates signify vital changes to the dependency. For this reason, version lag complements the time lag perfectly. For further understanding, it is agreed that a time lag of 20 days caused by a missed patch update is less severe than another lag of 8 days that was caused instead by a major update.

Moreover, upon refining the dataset in order to bypass the inaccuracies, the release frequency and technical lag values can be related. For reasons explained in each corresponding section, the Spearman's Rank Test statistic test was chosen for the correlation.

Preferably, the release frequency used for this question should be computed over a more relatively recent period, for instance, using the intermediate version releases happening within a client's prior and current releases, to better account for the update dynamics in the short term. Unfortunately, this was not achievable because there was no dependency data for earlier versions of many clients. Consequently, a different more general release frequency was utilized based on the complete set of dependency versions that were available since the client's first release date. This method, while capturing the more general update frequency of dependencies, does not account for the relative update frequency in the short term that may impact client technical lag.

4.5 Applying the Spearman's Rank Test

In this thesis, the Spearman analysis was applied in order to determine the relationships between the three main variables. Release frequency, breaking changes and technical lag. It was specifically chosen because it does not assume normal distribution of the data and it is ideal for examining monotonic relationships, which would be likely to appear. The test also focuses on ranks rather than raw values, which makes it even more suitable for the analysis.

For the first research question, the aim is to find out if an increase in the frequency of releases of a library is also likely to result in less breaking changes. A positive correlation coefficient would suggest that this is false, but a negative one would suggest that it is true, assuming that the p-value is smaller than 0.05.

In regard to the second research question, the goal is to determine whether dependency updates that contain more breaking changes and are used by clients tend to cause an increase in the client’s technical and version lag. A positive coefficient would confirm this hypothesis while a negative one would deny it, assuming the p-value is smaller than 0.05.

Lastly, in the case of the third research question, the test is used to analyze the impact that the release frequency of a dependency library has on the technical lag of the clients. A negative coefficient would suggest that the higher the release frequency of the dependency is, the lower the technical lag of the client will be. A positive coefficient would suggest the opposite. For the coefficient to be meaningful and statistically significant, the p-value must always be within the range of 0 and 0.05.

Chapter 5

Results

5.1 RQ1 Results

The Spearman's rank correlation was conducted in order to answer RQ1. The test returned a Spearman correlation coefficient of -0.197, with a p-value smaller than 0.05. This suggests a statistically significant but weak monotonic negative relationship between breaking changes and release frequency. Essentially this indicates that library versions that are updated more frequently usually introduce fewer breaking changes. However, further investigation would be needed in order to certify the validity of this claim, as the relation is not very strong. This can be confirmed by the correlation plot in Figure 7.

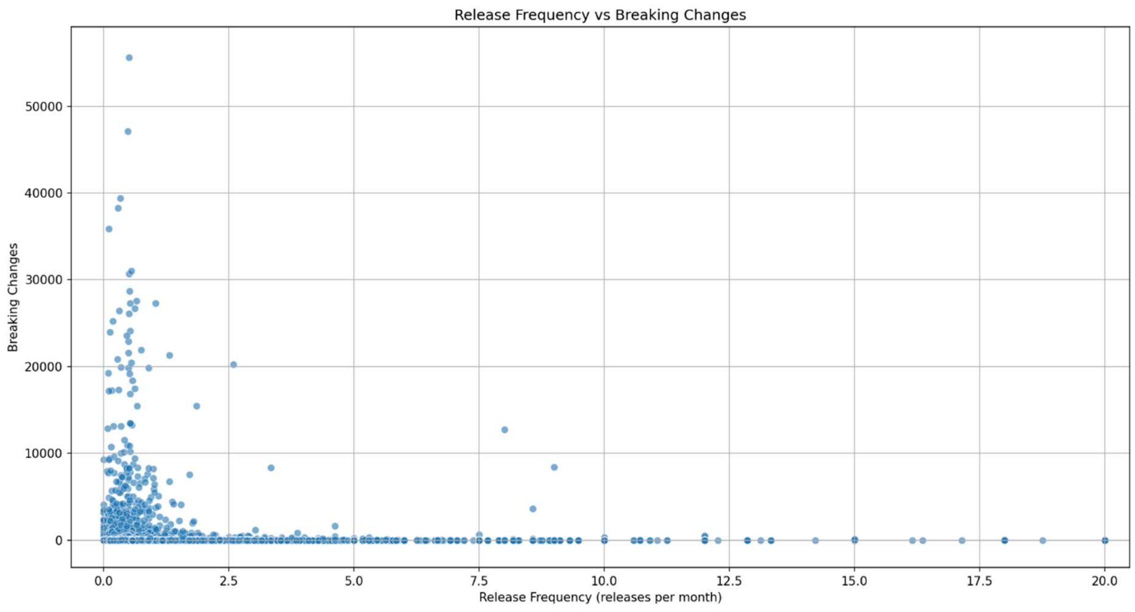


Figure 7

Breaking Changes between a library update vs Release Frequency of the library

5.2 RQ2 Results

The Spearman's rank correlation was conducted in order to answer RQ2, the relationship between dependency breaking changes and client technical lag. The test returned a Spearman correlation coefficient of 0.102, with a p-value smaller than 0.05. This suggests a statistically significant but fairly weak positive monotonic relationship between breaking changes and technical lag. Clients using dependency updates with more breaking changes tend to have higher technical lag, however the number of breaking changes which appear in the dependencies does not always accurately determine the amount of technical lag that the clients will end up with, as the increase in technical lag is not proportionate. This can be confirmed by the plot in Figure 8.

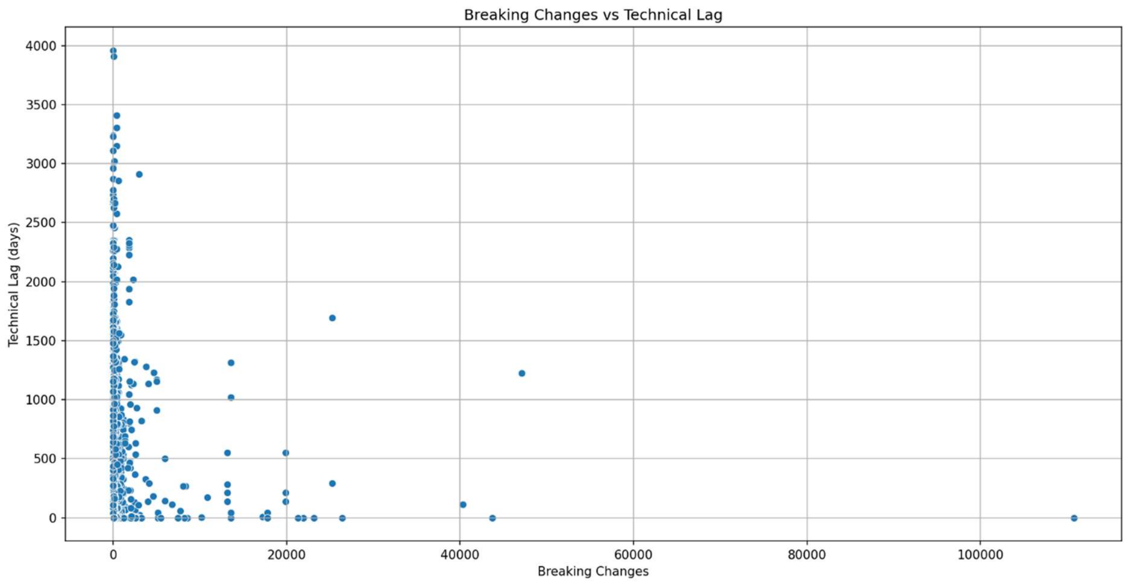


Figure 8

Breaking Changes between a library update vs Technical Lag of a client using the updated dependency

To further contextualize the relationship between dependency breaking changes and client outdatedness, another Spearman Correlation Test was applied on the dataset. In this case, the variables in question were the breaking changes and the version lag. The correlation of the breaking changes with the minor and major types of version lag proved to be the most interesting. The correlation between the dependency breaking changes and the minor version lag (Figure 9) returned a coefficient of 0.189 and a very small p-value, suggesting that clients that choose to skip multiple minor updates are likely to end up choosing a dependency version with a high number of breaking changes. At the same time, the correlation with the major version lag (Figure 10) returned a coefficient of -0.146 alongside a very small p-value again, implying that when more major updates are skipped, the client developers usually end up choosing a less severe – in terms of breaking changes – version of their dependency. Both the last two results are notable and their explanation will be discussed in the next Chapter.

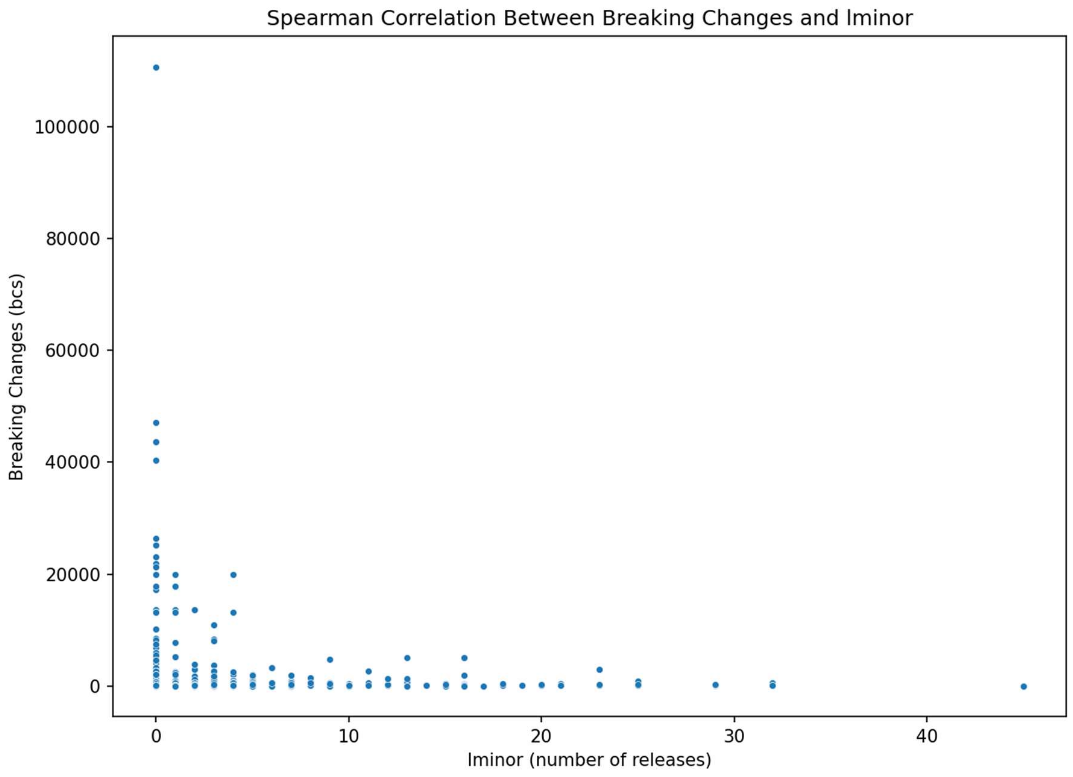


Figure 9

Breaking Changes between a library update vs Minor Version Lag of a client using the updated dependency

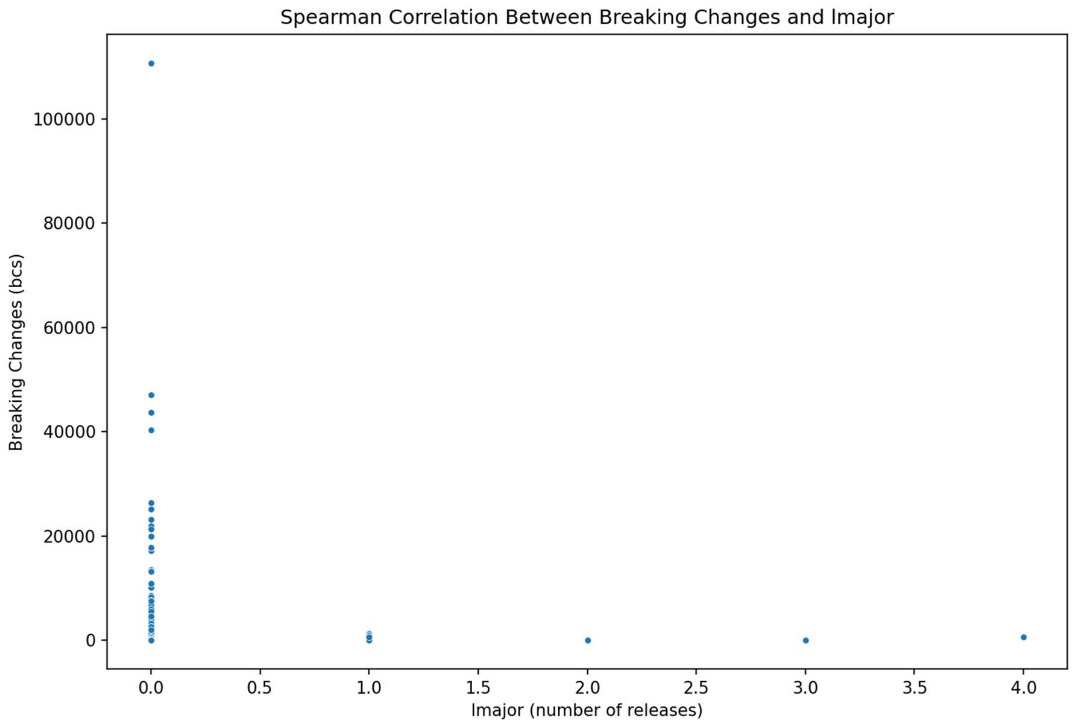


Figure 10

Breaking Changes between a library update vs Major Version Lag of a client using the updated dependency

5.3 RQ3 Results

The Spearman's rank correlation was conducted in order to answer RQ3, the relationship between dependency release frequency and client technical lag. The test returned the plot in Figure 11, alongside a correlation coefficient of 0.009 and a p-value of 0.599, which suggests that there is no statistically significant monotonic or linear relationship between the two variables, meaning that the frequency in which a dependency releases does not have a direct linear effect on the amount of technical lag that it causes a client version to obtain. This could be an indication of a more complex relationship between the two, which would require more factors in order to understand it.

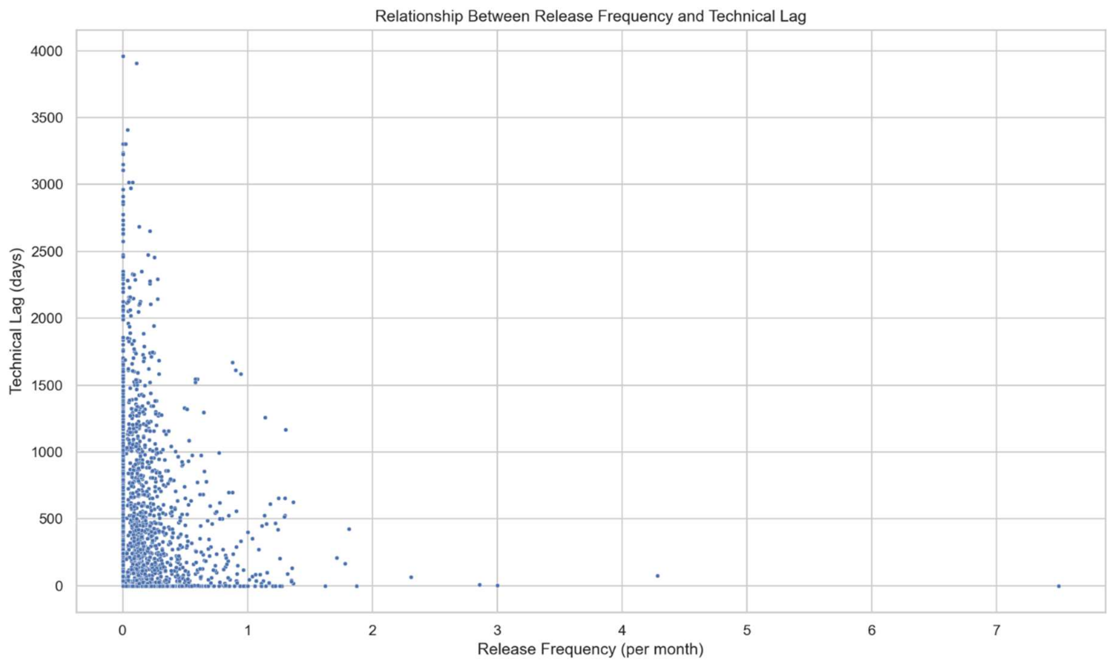


Figure 11

Technical Lag of a client using a dependency vs Release Frequency of the dependency

Chapter 6

Discussion

6.1 Discussion of the results

The results of the statistical analyses conducted for the research questions of this thesis have revealed some interesting and complex relationships between the three main variables.

Starting from the first question, we can understand that an increase of the release frequency of a library could be an indication of less breaking changes between its versions. However, it must be noted that their relationship is relatively weak in terms of the correlation coefficient. One reason for this might be that the developers may focus on constantly fixing minor bugs, which would increase release frequency and at the same time not introduce many breaking changes. On the other hand, the releases might be less frequent but more effective, in terms of dealing with more important issues of the code, which would demand an increase in breaking changes. These two scenarios could be cancelling each other out, and therefore prevent the relationship from being stronger. Moreover, the complexity of a dependency having to be compatible with another and the size of a library version update could complicate things even more. In conclusion, release frequency is an important indicator of breaking changes, but it is definitely not the only one.

As for the second question, the results of the analysis showed that there is a weak correlation between the number of breaking changes of a dependency update and the technical lag that it causes to its client. Updates that include more breaking changes sometimes tend to increase their client's technical lag. However, since the coefficient is not high, there may be other factors which could affect the relationship. The developers of the clients may delay their updates due to the fear of breaking changes, even if they are of high amounts numerically, as they might not want to take any risks. Another reason why the relationship is weak might be that the developers may choose to delay updates due to resource limitations, and not because of a high number of breaking changes. While breaking changes are an important factor of technical lag, ultimately the decision of the clients' developers changes based on their priorities. Furthermore, after a deeper analysis was applied using the types of version lag, the fact that developers often choose updates that introduce multiple breaking changes, after skipping many minor updates, suggests that breaking changes tend to be more common in scenarios where clients have skipped several minor versions. This pattern implies that updates following multiple minor releases could be more disruptive. On the other hand, the fact that developers often choose updates that introduce fewer breaking changes, after skipping many major updates indicates that they may be more cautious in their updating behavior, choosing to select their dependencies more carefully. These observations suggest that the types of versions that were skipped also play a role in the relationship between breaking changes and technical lag as well as in the developers' updating behavior.

In the case of the third question, the results are more complex and not straightforward. The Spearman analysis was unable to find a statistically significant direct correlation between the two variables, release frequency and technical lag. This could indicate that their relationship is non-linear and hides more complexity than expected, and thus it needs further investigation. Alternatively, the complexity of the results could be attributed to the way of calculating the release frequency, which would ideally be calculated as explained in Section 4.4. Due to the lack of availability of dependency data for earlier client versions, a more general release frequency measure was used, which may not fully reflect the short-term update pressure a client faced.

While frequent dependency updates may reduce breaking changes, they do not directly reduce clients' technical lag, as they are not the only factor influencing it. For a more complete view on the relationship between the three main variables, we would need to explore additional factors such as developer behavior using machine learning models.

6.2 Threats to Validity

6.2.1 Internal threats to validity

As mentioned in previous chapters, this thesis uses the dataset found in [4] and therefore any threats to validity that stand for the process of data collection of that study also stands for this one. Moreover, adding to that threat, it can be said that after fetching the release dates of the libraries that were relevant to this thesis, the internal threats were increased. It was evident that some dates were inaccurate, as there were common release dates amongst different versions of the same library. Therefore, any libraries that were connected to such cases had to be excluded from the filtered dataset that was used to answer the research questions.

Additionally, as mentioned in the Methodology and Discussion Chapters, the method used to calculate release frequency in RQ3 could introduce some threats to validity. Due to the lack of available dependency data for earlier versions of many clients, it was not possible to calculate a short-term release frequency. Instead, a more generalized release frequency was used, which may have weakened the relationship between release frequency and technical lag.

6.2.2 External threats to validity

The dataset that was used to answer the three research questions set cannot represent all of the libraries that have ever been created or uploaded on the internet. There are billions of libraries available on various repositories of several programming languages throughout the internet and generalizing the conclusion of this thesis for all the libraries internet-wide would be an external threat to validity.

6.2.3 Construct threats to validity

Technical lag was an important factor for the purposes of this thesis. However, as mentioned in Chapter 3, there are cases where a simple numeric value is not enough to measure how outdated a library is. This is the reason for the inclusion of the version lag measurement in the thesis, as it assists in forming a better picture for the outdatedness of a library and in parallel challenges a construct threat to validity.

6.2.4 Conclusion threats to validity

This thesis challenges any conclusion threats to validity by choosing to verify the results for each research question with the appropriate statistical tests based on the nature of each question and the variables that it requires. In Chapter 4, the reasoning behind the choices of statistical methods is briefly explained.

Chapter 7

Conclusion

7.1 Conclusion and Future Work

In conclusion, this thesis examined the relationships between dependency release frequency, breaking changes and client technical lag, using data from both the MCR and [4]. It addressed three research questions to better understand whether the frequency of releases affects the appearance of breaking changes and client lag. It also examined the effect that dependency updates which introduce breaking changes have on client lag.

The results revealed statistically significant but not very strong correlations for the first two RQs. They showed that an increase in release frequency tends to help decrease the number of breaking changes. At the same time, the thesis revealed that dependency updates that introduce a high number of breaking changes could lead to an increased technical lag of the clients that use them. Moreover, the results showed that the types of dependency updates which a client decides to skip also play their role in the connection of the two main variables of RQ2. However, no statistically significant linear relationship between release frequency and technical lag was found in RQ3, which could mean that their relationship is more complex and that it requires a non-linear analysis or a redefined release frequency variable to be fully grasped.

The fact that the findings did not reveal any strong correlations for the first two RQs, and also did not find any correlation at all for RQ3 suggests that there are multiple factors influencing the relationship of the three variables, such as developer behavior, project priorities, or risk management practices. This means that developer behavior cannot be easily predicted by linear analyses.

Future work for this thesis may include analyzing the three RQs through non-linear techniques in order to better understand the relationships which might be more complex than they seem to be. Moreover, a way to measure or quantify different developer behaviors based on several scenarios would help explain unexpected results throughout all the relationships of the three variables. Additionally, future work could improve this analysis by incorporating dependency data from earlier versions of clients, which could lead to a more accurate correlation between the two variables of RQ3.

Bibliography

[1] Bavota, Gabriele, et al. "The impact of API change-and fault-proneness on the user ratings of Android apps." *IEEE Transactions on Software Engineering* 41.4 (2014): 384–407.

[2] Decan, A., T. Mens and E. Constantinou. "On the Evolution of Technical Lag in the npm Package Dependency Network." *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid, Spain, 2018, pp. 404–414. doi: 10.1109/ICSME.2018.00050

[3] Li, Zengyang, Paris Avgeriou, and Peng Liang. "A systematic mapping study on technical debt and its management." *Journal of Systems and Software* 101 (2015): 193–220.

[4] Ochoa, L., Degueule, T., Falleri, JR. et al. "Breaking bad? Semantic versioning and impact of breaking changes in Maven Central." *Empirical Software Engineering* 27, 61 (2022). https://doi.org/10.1007/s10664-021-10052-y

[5] Raemaekers, S., A. van Deursen, and J. Visser. "Semantic versioning and impact of breaking changes in the Maven repository." *Journal of Systems and Software*, Volume 129, 2017, Pages 140–158. https://doi.org/10.1016/j.jss.2016.04.008

[6] Robbes, Romain, Mircea Lungu, and David Röthlisberger. "How do developers react to API deprecation? The case of a Smalltalk ecosystem." *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, 2012. https://doi.org/10.1145/2393596.2393662

[7] Stringer, J., A. Tahir, K. Blincoe and J. Dietrich. "Technical Lag of Dependencies in Major Package Managers." *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, Singapore, 2020, pp. 228–237. doi: 10.1109/APSEC51365.2020.00031.

[8] Xavier, L., A. Brito, A. Hora and M. T. Valente. "Historical and impact analysis of API breaking changes: A large-scale study." *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Klagenfurt, Austria, 2017, pp. 138–147. doi: 10.1109/SANER.2017.7884616

[9] Zerouali, A., E. Constantinou, T. Mens, G. Robles, and J. González-Barahona. "An Empirical Analysis of Technical Lag in npm Package Dependencies." (2018).

Appendix A

The GitHub link containing all related initial datasets and scripts to generate follow-up datasets:

https://github.com/cpapan02/CS401