# An Empirical Study of Version Conflicts in Maven-Based Java Projects

## Analyzing Developer Effort, Semantic Versioning Adherence and Resolution Strategies as Observed in Real-World Version Conflicts

**Valentin-Vlad Mihăilă**[1]

**Supervisors: Sebastian Proksch**[1]**, Cathrine Paulsen**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Valentin-Vlad Mihăilă
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Cathrine Paulsen, Georgios Iosifidis

An electronic version of this thesis is available at http://repository.tudelft.nl/.

## Abstract

Java projects often depend on third-party libraries to support development, but intensive reuse can lead to version conflicts, a common manifestation of dependency hell. This paper presents an empirical study of 124 GitHub pull requests from 85 Maven-based Java projects that addressed such version conflicts. We investigate the phenomenon from three perspectives: developer effort, the role of Semantic Versioning (SemVer) and conflict resolution strategies. Our analysis reveals that activity-based effort metrics are often confounded by unrelated changes and automation, suggesting the need for qualitative validation. Despite widespread use of SemVer syntax, 80% of observed runtime errors resulted from forward incompatibilities not covered by SemVer and even SemVer-compliant library versions occasionally broke backward-compatibility. The most common resolution strategy was library harmonization (67.7% of PRs), often achieved through version alignment techniques. These findings highlight the limitations of relying solely on versioning conventions and emphasize the importance of proactive tools and practices for managing dependencies. All datasets and analysis scripts are publicly available to support further research.


## 1 Introduction

The widespread adoption of open-source software development has led to a large collection of freely available software components (libraries) that can be used as building blocks for new projects [1]–[3]. Leveraging existing libraries can increase software quality, boost developer productivity and reduce development costs [2]. However, intensive use of third-party libraries may lead to complex dependency management issues (colloquially referred to as *dependency hell*), since developers often struggle to maintain a clear understanding of all software packages their project depends on [4].

One common class of dependency issues are version conflicts, which occur when a project depends on multiple versions of the same library. Such conflicts usually emerge when projects depend on third-party libraries which in turn depend on different versions of a shared library. When multiple versions of a library appear on the classpath of a Java project, typically only one version is loaded at runtime, while the others are shadowed [5]. This mismatch can potentially result in errors or unpredictable program behaviour at runtime, especially if the application depends on features that exist only in the versions that are not actually loaded.

Prior work has explored the manifestation and resolution patterns of general dependency conflicts in open-source software [6]–[8]. In particular, Wang et al.'s 2018 study [6] was the first to conduct an empirical study on dependency conflicts in open-source Java projects, applying a sound and reproducible methodology to retrieve documented occurrences of dependency conflicts. However, their dataset included only 39 cases of conflicts occurring in library versions and was targeted exclusively at the Apache software ecosystem. Although this ecosystem benefits from high-quality maintenance and documentation, the narrow scope limits the generalizability of their findings.

In this study, we replicated the methodology used by Wang et al. [6] and investigated version conflicts in a larger and more diverse sample of open-source projects. We carried out an empirical analysis of 124 Pull Requests (PRs) from GitHub that addressed version conflicts in 85 Java projects using the Maven build system. Specifically, our study examined version conflicts from three complementary angles:

**RQ1: How can we quantitatively measure developer effort spent resolving version conflicts?** Tracking the effort of developers in open-source software is inherently difficult due to challenges such as unstandardized documentation practices [9]. As a result, there is limited quantitative understanding of how version conflicts impact developers. To address this gap, we investigated four effort-related metrics derived from GitHub activity data and the analysis revealed that these metrics are not always reliable, as PRs often include unrelated changes or automation commands. These results suggest that purely quantitative measures may be insufficient on their own and highlight the need for qualitative methods to validate and contextualize such effort estimations.

**RQ2: To what extent does adherence to Semantic Versioning mitigate runtime errors caused by version conflicts?** Semantic Versioning (SemVer) aims to mitigate dependency hell by defining version rules that ensure backward-compatibility between patch and minor library updates. To the best of our knowledge, no prior work has investigated whether SemVer actually mitigates errors caused by version conflicts in practice. To address this gap, we analyzed version conflicts in 70 Java projects and found that compliance with SemVer syntax alone does not guarantee compatibility. In three concrete version conflict cases, patch or minor differences caused runtime errors due to backward-compatibility violations. Moreover, 80% of runtime errors were caused by forward incompatibilities (loaded versions missing functionality referenced through newer versions), where SemVer offers no protection. These findings suggest that SemVer's effectiveness is limited in practice and relying solely on version numbers is often insufficient to ensure compatibility.

**RQ3: What resolution strategies do developers use to fix version conflicts?** While Wang et al. [6] identified several resolution patterns for addressing version conflicts in Maven, the small size and ecosystem-specific scope of their dataset may limit the broader applicability of their findings. To extend their work, we analyzed 124 PRs and identified five main resolution categories. Among these, harmonization (i.e., aligning) of library versions was the most frequently applied strategy, with developers favouring either local adjustments (e.g., modifying direct dependencies) or centralized control mechanisms (e.g., via the `dependencyManagement` section). In addition, we observed a smaller but noteworthy use of proactive conflict prevention strategies that offer promising opportunities to detect and avoid conflicts earlier in the development process.

Ultimately, our findings provide real-world insights into version conflicts as a common manifestation of dependency

hell and aim to inform the development of more effective dependency management tools and practices. By helping developers better manage project dependencies, we hope to support more reliable and maintainable software. To facilitate future research in this area, we have made our datasets and analysis scripts publicly available on GitHub [10].

## 2 Background and Related Work

This section introduces key concepts and prior work relevant to understand version conflicts in Maven-based Java projects. We begin with a real-world example, followed by an overview of Maven's dependency mechanism and the role of Semantic Versioning. We conclude with related studies on dependency conflicts, developer effort and resolution practices.

### 2.1 Real-World Version Conflict Example

To illustrate a real-world case of a version conflict, consider an issue from the GitHub repository of Apache Pulsar [11] that resulted in a runtime failure when executing a shell command. As reported in PR apache/pulsar#16775 [12], a dependency of the project (`JCommander 1.78`) overshadowed an older release of the same library (`JCommander 1.48`), which was required by another dependency of the project (`BookKeeper 4.12.0`). The newer `JCommander` version had removed a method still used by `BookKeeper`, triggering a `NoSuchMethodError` at runtime (Figure 1). Fortunately, the issue was solved by upgrading `BookKeeper` to version `4.12.1`, which also uses `JCommander 1.78`.

```java
public class JCommander { // 1.48
  /* ... */
  public void usage(StringBuilder out) {
    usage(out, "");
  }

  public void usage() {
    StringBuilder sb = new StringBuilder();
    usage(sb);
    getConsole().println(sb.toString());
  }
}
```

(a)

```java
public class JCommander { // 1.78
  /* ... */
  private IUsageFormatter usageFormatter =
      new DefaultUsageFormatter(this);

  public void usage() {
    StringBuilder sb = new StringBuilder();
    usageFormatter.usage(sb);
    getConsole().println(sb.toString());
  }
}
```

(b)

```
$ bin/bookkeeper shell cookie_update -h
JMX enabled by default
Exception in thread "main" java.lang.NoSuchMethodError:
    com.beust.jcommander.JCommander.usage(Ljava/lang/StringBuilder;)
    at org.apache.bookkeeper.tools.framework.Cli.setupCli(Cli.java:142)
    at org.apache.bookkeeper.tools.framework.Cli.<init>(Cli.java:53)
    at org.apache.bookkeeper.tools.framework.Cli.runCli(Cli.java:243)
    at org.apache.bookkeeper.tools.common.BKCommand.apply(BKCommand.java:64)
    ...
```

(c)

Figure 1: Example of a version conflict causing a runtime error due to version incompatibilities. The method `usage(StringBuilder)` of the `JCommander` class in `JCommander 1.48` (a) was removed in favour of a helper object in `JCommander 1.78` (b), raising `NoSuchMethodError` when the method was invoked at runtime (c).

### 2.2 Maven Dependency Resolution Model

Maven is an automated build system for JVM-based languages (such as Java, Kotlin or Scala) that manages the entire build cycle of software projects. To use Maven, a Project Object Model (POM) is created in a `pom.xml` file, which specifies the project structure and third-party libraries (artifacts) that the project depends on. Dependencies declared in the POM are referred to as *direct dependencies* and they often have their own dependencies, which are called *transitive dependencies*. For example, consider a Maven project that uses the Spring Framework library [13], which requires transitive dependencies such as Spring Core, Spring Context and Spring Security. Maven resolves these transitive dependencies automatically, thus relieving developers from the complexity of manually adding all required dependencies [14].

By default, Maven dependencies are specified using fixed version numbers, a practice known as version pinning. While this approach ensures reproducible builds, it may also lead to version conflicts. In particular, when commonly used libraries (e.g., Guava, Apache Commons, Jackson) are restricted to specific versions in the dependency graphs of other libraries, users are exposed to a higher likelihood of version conflicts when relying on these libraries [7]. To mediate such conflicts, Maven applies a "nearest wins" strategy, selecting the version that is closest to the root of the project's dependency tree [14]. In case multiple versions are located at the same depth, Maven chooses the one that is declared first.

However, issues may arise if the selected library version lacks functionality required by project components that rely on different, incompatible versions of the same library. Such version conflicts can lead to common runtime errors such as `NoSuchMethodError` and `NoClassDefFoundError`, which indicate missing method or class definitions. In order to guarantee the version of a library, Maven recommends declaring it explicitly as a direct dependency in the project's POM or controlling the version centrally in the dedicated `dependencyManagement` section of the POM [14].

To aid developers detect version conflicts in their project's dependencies, Maven offers the Maven Dependency Plugin [15] which can be configured to show warnings of common dependency issues when a project is built. Running the Maven `dependency:tree` goal with the `verbose` flag enabled generates a serialized tree of the project's dependencies, specifying which ones were omitted due to version conflicts [16]. Figure 2 shows an example dependency tree where multiple versions of the Guava library are introduced transitively. Maven resolves the conflict by selecting version `21.0` (the nearest to the root) and omits versions deeper in the hierarchy (`19.0`, `17.0`). Additionally, identical versions from different paths are marked as duplicates.

```
me.lucko:bungeeguard-sponge:1.2-SNAPSHOT
 +- me.lucko:bungeeguard-backend:1.2-SNAPSHOT
 \- org.spongepowered:spongeapi:7.3.0
    +- com.google.guava:guava:21.0
    +- com.google.inject:guice:4.1.0
    |  \- (com.google.guava:guava:19.0 - omitted for conflict with 21.0)
    +- org.spongepowered:plugin-meta:0.4.1
    |  \- (com.google.guava:guava:17.0 - omitted for conflict with 21.0)
    \- org.spongepowered:configurate-hocon:3.7.1
       \- org.spongepowered:configurate-core:3.7.1
          \- (com.google.guava:guava:21.0 - omitted for duplicate)
```

Figure 2: Simplified dependency tree with two Guava version conflicts and one duplicate in the POM of `bungeeguard-sponge` before the changes in lucko/BungeeGuard#75 [17].

However, Maven cannot distinguish between compatible and harmful version conflicts (i.e., ones that will lead to runtime failures) [6]. As the number of a project's dependencies grows, the dependency tree can also quickly become very large and difficult to understand. Therefore, such warnings

may be overlooked by developers, potentially leading to serious consequences in practice.

### 2.3 Semantic Versioning

Semantic Versioning (SemVer) has been proposed as a partial solution to the dependency hell problem, encoding compatibility promises in the version numbers of software packages [18]. Introduced by GitHub co-founder Tom Preston-Werner, SemVer defines a three-digit versioning scheme (`MAJOR.MINOR.PATCH`), each indicating a different level of change:

- `MAJOR`: API changes breaking backward-compatibility;
- `MINOR`: backward-compatible changes or new features;
- `PATCH`: only backward-compatible bug fixes allowed.

According to this scheme, version conflicts that differ in the `PATCH` or `MINOR` parts should be backward-compatible, meaning that newer patch or minor releases should cover previous functionality. For instance, if a project depends on version `2.1.0` of a library, it should continue to function correctly when a newer patch version like `2.1.3` or a newer minor version like `2.2.0` is loaded. However, because adherence to SemVer is not strictly enforced, library developers may misuse the versioning standards, introducing unexpected breaking changes despite compatible version numbers. To mitigate such issues, platforms like GitHub and Maven Central strongly encourage developers to follow SemVer when making new releases [19].

### 2.4 Related Work

Wang et al. [6] conducted a large-scale analysis of 2,289 Java projects on GitHub and found that 63.65% contained different versions of the same library. In a more focused study of 135 dependency conflict issues across 71 Java projects hosted on the Apache ecosystem, they observed that 39 issues were caused by distinct library versions present in the projects. The remaining issues were attributed to class conflicts either among libraries (90 cases) or between libraries and the host project (6 cases). They also examined common strategies to resolve version conflicts, including harmonizing library versions, reordering dependencies on the classpath and using the Maven Shade Plugin [16] to isolate conflicting libraries. Additionally, the authors analyzed developer discussions in issue reports on the Jira tracking system to identify factors impacting the effort required for resolving dependency issues.

To better understand the challenges developers face in dependency management, Pashchenko et al. [4] conducted 25 semi-structured interviews with developers from both small-medium and large enterprises. Their findings revealed that many developers are reluctant to update dependencies due to the risk of breaking changes and the lack of resources to address them. A key concern was the presence of hidden breaking changes in transitive dependencies, which affected the developers' confidence in updating dependencies.

The developers' reluctance to update dependencies is further complicated by library release practices and the structure of large artifact repositories (e.g., Maven Central). Soto-Valero et al. [20] studied the co-existence of multiple library versions in Maven Central, reporting that 40% of libraries have at least two actively used versions. Similarly, Raemaekers et al. [21] analyzed the use of SemVer in Maven Central libraries and found that SemVer is frequently misused, with breaking changes sometimes introduced even in minor version updates.

## 3 Data Collection

To answer the three research questions outlined in Section 1, we conducted an empirical study on documented occurrences of version conflicts in open-source projects. This section describes the motivation behind the chosen methodology and then details the three steps taken to compile the dataset.

### 3.1 Motivation

The empirical study targeted open-source Java projects hosted on GitHub that use Maven as a build system. As the world's largest source code hosting platform, GitHub hosts over 250 million open-source projects which received over one billion contributions in 2024 alone [22]. The Maven ecosystem was chosen due to its (i) *large scale* (indexing over 56 million packages as of May 2025 [23]), (ii) *widespread adoption* (more than 76% of the JVM developers used the Maven build system in 2021 [24]) and (iii) *explicit versioning and dependency resolution mechanisms* (as discussed in Subsection 2.2). These three characteristics collectively increase the likelihood and impact of version conflicts, making the Maven ecosystem a compelling case for our study.

The methodology closely follows the approach proposed by Wang et al. [6], which was replicated in a subsequent investigation of version conflicts in Python projects [7]. In both studies, the authors began by compiling a corpus of open-source projects, then applied keyword-based searches to identify relevant issue reports using terms typically associated with dependency conflicts. Since the automated process could yield duplicates or irrelevant results, all identified candidates were manually reviewed to ensure their validity. Following this methodology, our data collection was carried out in three steps, which are detailed in the following subsections and summarized in Figure 3.

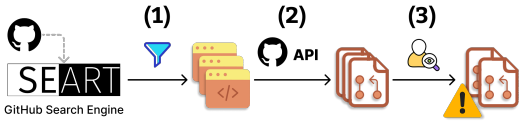



Figure 3: Overview of the three-step data collection process: (1) compile a corpus of open-source projects, (2) perform keyword-based searches to identify potential version conflict reports and (3) manually review results to ensure relevance and remove duplicates.

### 3.2 Project Selection

First, we retrieved an initial set of 5,919 Java projects using the SEART GitHub Search Engine [25] based on a snapshot from April 27, 2025. This tool allows for selecting GitHub repositories based on custom filter criteria. Our selection was based on the following four criteria: (i) the repository uses the Java language, (ii) it was created between April 2015 and

April 2025, (iii) it has received at least 50 stars (often used as popularity metric by other studies [6]–[8]) and (iv) it has at least 50 total issues (used as an indicator of maintainability by other studies [7]). The tool returned 6,271 results, from which we excluded invalid entries that were no longer accessible and duplicates such as redirected repositories.

### 3.3 Version Conflict Identification

Second, 196 Pull Requests (PRs) were identified using the GitHub REST API [26] based on a keyword search using phrases and errors related to version conflict issues. PRs contain changes intended to address specific problems or introduce improvements in open-source software, thus analyzing PR reports provides insight into the problems developers encountered and how they were fixed [27]. To feasibly detect PRs resolving version conflicts in the 5,919 repositories, we used two inclusion and two exclusion criteria. We included PRs that (i) were merged (whose changes were accepted and integrated into another branch, typically the main or development branch) and (ii) contained at least one of the following three keywords: "version conflict*", "library conflict*" and "`NoSuchMethodError`". We excluded PRs that (i) were submitted by automated bots or (ii) did not modify the POM, since resolving version conflicts in Maven generally requires modifying the dependency configuration [6].

### 3.4 Manual Inspection

Third, to ensure the validity of the automated search results, we conducted a manual inspection of the 196 PRs identified in the previous step. During this step, we checked whether the PR actually resolves version conflicts occurring in Maven dependencies. We removed false positives (e.g., errors caused by Maven or Java runtime versions, or cases where the keyword search matched references of version conflicts in documentation files), as well as duplicates (e.g., GitHub cherry-picks, which apply a commit to different branches). The manual review resulted in a final dataset of 124 PRs spanning 85 repositories, which served as the basis for answering all three research questions. Table 1 presents five key metrics of these repositories, illustrating the diversity of the chosen sample.

Table 1: Demographics of the 85 repositories covered by the final sample of 124 PRs ([1]KLOC denotes thousand lines of code).

| Metric | Min | Max | Median |
|---|---|---|---|
| KLOC[1] | 1.29 | 6,041.45 | 187.21 |
| Commits | 46 | 58,564 | 2,698 |
| Pull Requests | 17 | 49,841 | 1,679 |
| Stars | 54 | 31,254 | 499 |
| Contributors | 8 | 406 | 73 |

## 4 Developer Effort (*RQ1*)

To better understand the level of developer effort involved in resolving version conflicts, we investigated four metrics derived from publicly available GitHub data. These metrics quantify different aspects of effort, such as communication, duration and code modification. This section first describes the methodology used to compute these metrics across the sample of 124 PRs, followed by a presentation of the results.

### 4.1 Methodology

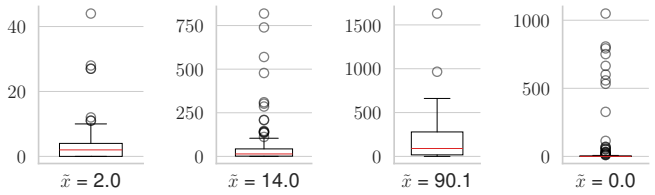
We evaluated developer effort in PRs by extracting relevant metadata from the GitHub API to compute four key metrics:

1. **Comments**: the number of comments posted in the PR (as used by Wang et al. [6]). Comments are generally used by developers to clarify implementation details or request changes, an important step when solving bugs. To differentiate actual human discussions from comments generated by bots and commands used to trigger automation tasks, comments were classified as *impure* if they were created by usernames containing the "[bot]" tag, or if they begin with "run" or "rerun";
2. **Merge Time**: the time interval from when the PR was submitted until it was merged. This metric indicates the total duration from the initial submission of the proposed changes, undergoing code reviews and potential modifications, until the fix is finally accepted and integrated into the main or development branch;
3. **Detection to Resolution Time**: the time span between the discovery of the version conflict (as indicated through related issues explicitly linked in the PR) and its resolution (the merge time), also used in [6]. This captures the time length developers might have been impacted by the issue and is indicative of how quickly version conflicts are diagnosed and addressed in practice;
4. **Java Line Changes**: the number of lines added or removed in Java source code by the PR changes. When modifying the dependency configuration is not enough to resolve version conflicts, developers might need to make non-trivial refactorings in the source code.

To compare the effort involved in resolving version conflicts with typical development work, we analyzed the activity of each PR in the dataset relative to other merged PRs from the same repository. We applied z-score normalization $(x - \mu)/\sigma$ for the merge times and comment counts to evaluate how they deviate from average repository activity. For comment counts, we examined a subset of 85 PRs from 69 repositories. This subset was selected due to GitHub's rate limit of 5,000 API requests per hour, which made it unfeasible to process the remaining 16 larger repositories.

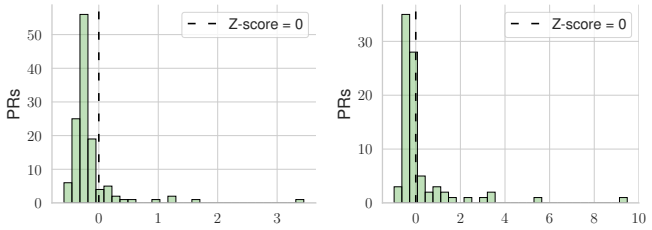
### 4.2 Results

The distribution of the four developer effort metrics is shown in Figure 4, with median values reported due to the presence of outliers. Most PRs (105 out of 124; 84.7%) contain between 0 and 5 comments (Figure 4a). Based on the impure comment detection method described earlier, 66 comments (15.4%) were identified as impure, although excluding impure comments did not have a significant impact on the overall distribution shown in Figure 4a. The median time to merge a PR was 14 hours (Figure 4b) and for 29 PRs that were linked to an issue, the median time from issue detection to resolution was 90.1 hours (Figure 4c). Using Spearman's rank correlation ($\rho$), we observed a moderate correlation between merge time and the number of comments ($\rho = 0.38$, $n = 124$), as well as between merge time and detection to resolution time ($\rho = 0.52$, $n = 29$).




(a) Comments (b) Merge Time (h) (c) Detection to Resolution Time (h) (d) Java Line Changes

Figure 4: Distributions of developer effort metrics derived from GitHub PR activity. (c) is based on a subset of $n = 29$ PRs with linked issues.




(a) Merge Time (z-score) (b) Comments (z-score)

Figure 5: Distributions of z-score normalized developer effort metrics, computed relative to all merged PRs within the same repository. (b) is based on a subset of $n = 85$ PRs where repository averages could be feasibly computed.

In terms of line changes, 89 PRs (71.8%) did not include any changes to Java source code (Figure 4d). Additionally, 34 PRs (27.4%) involved at most 5 lines of overall changes, considering additions and removals in the Git diff of all files.

The effort analysis based on the normalized merge time revealed that 107 PRs (86.3%) had a z-score below 0, indicating merge times shorter than average (Figure 5a). Within the smaller subset of 85 PRs, 63 PRs (74.1%) also had a negative z-score for the number of comments (Figure 5b).

# 5 SemVer Adherence (*RQ2*)

To assess how adherence to SemVer mitigates failures caused by version conflicts in practice, we analyzed version conflicts in 85 Java projects covered by our PR dataset. The identified conflicts were categorized based on their version differences and a subset of these was manually inspected to identify cases where violations of SemVer led to runtime errors. This section first describes the methodology used to detect and classify conflicts, followed by a presentation of key results.

## 5.1 Methodology

We used the Maven `dependency:tree` goal (as described in Subsection 2.2) to detect and categorize all version conflicts present in the sample of 85 repositories. To minimize potential bias, we selected only one PR per repository, investigating the state before the changes introduced by the selected PR. Since many repositories contain multiple modules (subprojects) with different POM configurations, we generated the dependency trees for all modules defined in the central POM file, which is located at the root of the project.

Within each module, every pair of conflicting versions that conformed to SemVer was assigned a **semantic difference**, classifying differences between versions as *Major*, *Minor*, *Patch* or *Other* (the latter including optional pre-release or build metadata labels). To parse dependency versions reliably according to the SemVer specification, we used the `semver` Python package [28], which supports versions with missing minor or patch parts (e.g., `1.3`).

To evaluate the reliability of SemVer compatibility guarantees in mitigating failures caused by version conflicts, we manually investigated PRs where the tree generation was successful to identify instances where developers explicitly reported software breakages due to version conflicts. For each identified case, we recorded the type of error encountered, the root cause dependency along with its two conflicting versions and the direction of the conflict (i.e., whether the loaded version was older or newer). The direction is particularly relevant, as newer patch and minor versions are expected to maintain backward compatibility under the SemVer specification.

## 5.2 Results

The dependency tree generation succeeded for 70 PRs, resulting in the detection of 52,417 version conflicts, 87.1% of which adhered to the SemVer syntax (Table 2). The remaining 15 PRs could not be processed due to the absence of a central POM (10 PRs), unresolved artifacts (3 PRs) or Maven compiler errors (2 PRs).

Table 2: Summary of semantic differences between conflicting versions across 2,199 affected modules in 70 GitHub repositories. Per-module maximum, median and mean counts are reported.

| Semantic Difference | Max | Median | Mean | Total | % |
|---|---|---|---|---|---|
| Major | 143 | 1 | 3.57 | 7,854 | 15.0% |
| Minor | 233 | 4 | 11.79 | 25,937 | 49.5% |
| Patch | 139 | 1 | 5.38 | 11,831 | 22.6% |
| Other | 7 | 0 | 0.02 | 38 | <0.1% |
| Invalid SemVer | 93 | 0 | 3.07 | 6,757 | 12.9% |
| *All* | *516* | *7* | *23.84* | *52,417* | *100%* |

In the sample of 70 PRs, we found 35 runtime errors that were directly caused by version conflicts, involving 8 major, 17 minor and 10 patch differences, as listed in Table 3. In 7 of the 35 cases, the error was caused by backward incompatibility (i.e., loading a newer library version which lacked functionality required through an older version). Notably, 3 of the 7 instances are considered direct violations of SemVer's backward compatibility rules (highlighted in red in Table 3), as `NoSuchMethodError` resulted from method incompatibilities when the newer minor or patch version was selected.

# 6 Resolution Strategies (*RQ3*)

To investigate how developers resolve version conflicts in Maven, we analyzed the sample of 124 PRs and identified common resolution patterns, which we grouped into five main categories. This section briefly describes the methodology used to identify and classify resolution strategies, and then presents the observed strategies.

Table 3: List of PRs with reported runtime errors that were caused by version conflicts. The versions in bold indicate cases where a newer dependency version was loaded which failed to provide functionality expected by another component of the project. The PRs highlighted in red are direct violations of SemVer's backward compatibility rules.

| # | PR Link | Error Type | Root Cause Dependency | Version Required | Version Loaded | Semantic Difference |
|---|---|---|---|---|---|---|
| 1 | https://github.com/Alfresco/alfresco-community-repo/pull/3063 | NoSuchMethodError | org.apache.logging.log4j:log4j-api | 2.31.1 | 2.24.2 | MINOR |
| 2 | https://github.com/alibaba/nacos/pull/10170 | NoSuchMethodError | org.apache.httpcomponents:httpasyncclient | 4.1.5 | 4.1.3 | PATCH |
| 3 | https://github.com/alldatacenter/alldata/pull/595 | NoSuchMethodError | net.java.dev.jna:jna | 4.1 | **5.8** | MAJOR |
| 4 | https://github.com/apache/amoro/pull/2561 | NoSuchMethodError | org.apache.orc:orc-core | 1.9.1 | 1.7.2 | MINOR |
| 5 | https://github.com/apache/camel-k-runtime/pull/1053 | NoSuchMethodError | ch.qos.logback:logback-classic | >1.3.0 | 1.2.11 | MINOR |
| 6 | https://github.com/apache/camel-quarkus/pull/736 | NoSuchMethodError | xml-apis:xml-apis | 1.4.01 | 1.3.04 | MINOR |
| 7 | https://github.com/apache/dolphinscheduler/pull/14561 | NoSuchMethodError | io.fabric8:kubernetes-client | 5.10.2 | **6.0.0** | MAJOR |
| 8 | https://github.com/apache/hop/pull/4606 | NoSuchMethodError | org.codehaus.woodstox:stax2-api | 4.2.1 | 3.1.4 | MAJOR |
| 9 | https://github.com/apache/incubator-hugegraph/pull/2339 | NoSuchMethodError | org.apache.logging.log4j:log4j-api | 2.17.2 | 2.17.0 | PATCH |
| 10 | https://github.com/apache/incubator-seata-samples/pull/171 | NoSuchMethodError | org.springframework:spring-web | 5.1.5 | 5.1.3 | PATCH |
| 11 | https://github.com/apache/pulsar/pull/20070 | NoSuchMethodError | io.vertx:vertx-core | 4.3.2 | 3.9.8 | MAJOR |
| 12 | https://github.com/apache/seatunnel/pull/2589 | NoSuchMethodError | commons-io:commons-io | 1.3.2 | **2.11.0** | MAJOR |
| 13 | https://github.com/apache/shardingsphere/pull/7299 | NoSuchMethodError | io.netty:netty-buffer | 4.1.45 | 4.1.42 | PATCH |
| 14 | https://github.com/confluentinc/ksql/pull/10033 | NoSuchMethodError | io.netty:netty-handler | 4.1.87 | 4.1.86 | PATCH |
| 15 | https://github.com/eclipse-hawkbit/hawkbit/pull/158 | NoSuchMethodError | com.vaadin:vaadin-shared | 7.6.5 | 7.6.3 | PATCH |
| 16 | https://github.com/eclipse-sw360/sw360/pull/459 | NoSuchMethodError | org.apache.commons:commons-lang3 | 3.8.1 | 3.1 | MINOR |
| 17 | https://github.com/google/caliper/pull/421 | NoSuchMethodError | com.squareup:javapoet | 1.11.1 | 1.9.0 | MINOR |
| 18 | https://github.com/googleapis/java-storage/pull/2680 | NoSuchMethodError | org.junit.platform:junit-platform-commons | 1.11.0 | 1.10.3 | MINOR |
| 19 | https://github.com/GoogleCloudPlatform/cloud-spanner-r2dbc/pull/541 | NoSuchMethodError | org.springframework.data:spring-data-commons | 2.7.0 | 2.6.4 | MINOR |
| 20 | https://github.com/GoogleCloudPlatform/DataflowTemplates/pull/1838 | NoSuchMethodError | ch.qos.logback:logback-core | 1.2.13 | 1.2.10 | PATCH |
| 21 | https://github.com/GoogleCloudPlatform/spring-cloud-gcp/pull/3146 | NoSuchMethodError | io.opentelemetry:opentelemetry-api | 1.40.0 | 1.37.0 | MINOR |
| 22 | https://github.com/hiero-ledger/hiero-mirror-node/pull/1252 | NoSuchMethodError | io.grpc:grpc-netty-shaded | 1.31.1 | 1.30.2 | MINOR |
| 23 | https://github.com/JanusGraph/janusgraph/pull/3764 | NoSuchMethodError | org.yaml:snakeyaml | 1.32 | **2.0** | MAJOR |
| 24 | https://github.com/openmessaging/benchmark/pull/392 | NoSuchMethodError | com.google.protobuf:protobuf-java | 3.21.5 | 3.9.1 | MINOR |
| 25 | https://github.com/opentracing-contrib/java-specialagent/pull/47 | NoSuchMethodError | javax.servlet:javax.servlet-api | 3.1.0 | 3.0.1 | MINOR |
| 26 | https://github.com/openzipkin/zipkin-gcp/pull/72 | NoSuchMethodError | com.google.guava:guava | 20.0 | 18.0 | MAJOR |
| 27 | https://github.com/powsybl/powsybl-core/pull/1471 | NoSuchMethodError | ch.qos.logback:logback-core | 1.2.3 | 1.1.8 | MINOR |
| 28 | https://github.com/sofastack/sofa-jraft/pull/1073 | NoSuchMethodError | com.caucho:hessian | 4.0.3 | **4.0.63** | PATCH |
| 29 | https://github.com/spring-cloud/spring-cloud-stream/pull/2489 | NoSuchMethodError | org.apache.kafka:kafka_2.13 | 3.1.1 | 3.0.0 | MINOR |
| 30 | https://github.com/streamnative/mop/pull/1406 | NoSuchMethodError | com.fasterxml.jackson.core:jackson-databind | 2.12.2 | **2.16.2** | MINOR |
| 31 | https://github.com/thingsboard/thingsboard/pull/5027 | NoSuchMethodError | io.netty:netty-buffer | 4.1.66 | 4.1.45 | PATCH |
| 32 | https://github.com/WesJD/AnvilGUI/pull/222 | NoSuchMethodError | org.spigotmc:spigot | 1.19 | **1.19.1** | PATCH |
| 33 | https://github.com/Consensys/tessera/pull/736 | NoSuchFieldError | org.bouncycastle:bcprov-jdk15on | 1.61 | 1.59 | MINOR |
| 34 | https://github.com/jenkinsci/github-checks-plugin/pull/26 | NoClassDefFoundError | com.google.guava:guava | 29.0-jre | 11.0.1 | MAJOR |
| 35 | https://github.com/OpenAPITools/openapi-generator/pull/7102 | Internal Server Error | com.fasterxml.jackson.core:jackson-annotations | 2.10.x | 2.9.0 | MINOR |

## 6.1 Methodology

Resolution strategies were identified through a manual review of each PR, examining developer discussions and PR file changes. We extracted specific strategies taken by developers to resolve version conflicts and then grouped them into broader categories based on shared patterns and goals. It is important to note that 19 PRs (15.3%) involved a combination of two or more strategies and in these cases each strategy was counted individually.

## 6.2 Results

We identified five categories of common resolution strategies that addressed version conflicts in 118 PRs (95.2% of the sample). The distribution of these categories is shown in Figure 6 and each category is described below.

**I. Controlling dependency versions locally** (47 out of 124 PRs). Developers often resolved version conflicts by explicitly adding direct dependencies to override the version selected through Maven's "nearest wins" mechanism (Subsection 2.2). In other cases, developers modified the version through an already existing direct dependency (either upgraded or downgraded) to meet the version requirements of other project components. A similar strategy was adjusting the version of a direct dependency that introduced the conflict transitively. For example, in citrusframework/yaks#39 [29], a mismatch between Spring 4.x and 5.x dependencies was resolved by both adding direct Spring dependencies and upgrading one of their parent dependencies.

**II. Managing dependency versions centrally** (44 out of 124 PRs). Several PRs resolved version conflicts by including the desired version in the `dependencyManagement` section, used to enforce a consistent version across multiple project components. In fewer cases, version alignment was achieved using BOM (Bill of Materials) POMs, which define compatible sets of versions for interdependent libraries [14]. For example, in googleapis/java-storage#2680 [30], a test failure caused by conflicting JUnit versions was resolved by importing the `junit-bom`, following the official JUnit recommendation [31].

**III. Excluding transitive dependencies** (27 out of 124 PRs). In 24 PRs, developers used the `<exclusion>` element to remove transitive dependencies which introduced conflicts, following Maven's recommendation to prevent unwanted dependencies from appearing on the project's classpath [32]. In 7 of these cases, the excluded dependency was then re-added as a direct dependency to ensure the correct version was used. For instance, in langchain4j/langchain4j#1508 [33], the author applied this two-step approach, describing that it helps "avoid version divergence in the project".

**IV. Removing or replacing dependencies** (12 out of 124 PRs). In some cases, developers removed the direct conflicting dependency, choosing to rely on a transitive version instead. Others resolved the conflict by replacing the conflicting dependency with another dependency, which sometimes required additional source code modifications. For example, in stargate/stargate#1253 [34], the author mentioned that in order to avoid Guava version conflicts, they replaced the "uses of plain Guava with either [the] shaded version or (in some cases) JDK 8 methods".

**V. Shading dependencies** (9 out of 124 PRs). When two conflicting versions were required at the same time (e.g., due to incompatible features), developers used the Maven Shade Plugin to relocate one of the versions within the *Uber JAR*, which packages the project dependencies in the final artifact. For example, in apache/iotdb#9788 [35], the author shaded all Thrift artifacts used in their project to allow their project and Spark dependencies to each use their desired Thrift versions without conflict.


Figure 6: Distribution of resolution strategies used by developers to address version conflicts in Maven-based Java projects. In cases where multiple strategies were used in a single PR (15.3% of the sample), each strategy was counted separately.

# 7 Discussion

This section discusses the key findings across all three research questions, reflecting on their implications for different stakeholders, comparing them with related work and suggesting potential directions for future research.

## 7.1 Limitations of PR Effort Metrics

**Noisiness Caused by Unrelated Changes.** In this study, we explored GitHub PR activity as an indicator for developer effort (Figure 4) and our findings suggest that the four proposed metrics are not always reliable, primarily because version conflict fixes are often embedded within broader development tasks (e.g., integrating new features, code refactoring), rather than addressed in isolation. As a result, unrelated changes in the PR can introduce noise that affects the accuracy of these effort estimates.

**Inaccurate Reflection of Human Effort.** While a higher merge time may intuitively imply more effort, it does not necessarily reflect continuous work, as merges can be delayed due to low urgency or scheduling constraints. For example, in matsim-org/matsim-libs#1291 [36] the merge was postponed until after the holiday season to avoid potential build failures, thus inflating the apparent effort.

Similarly, associating comment counts with human effort also has several limitations. It overlooks discussions held outside of GitHub and includes unrelated comments such as those used to trigger CI pipelines or test coverage tools. While the method described in Subsection 4.1 successfully flagged 15.4% of unrelated comments, a closer manual inspection revealed that not all such comments were detected, often due to inconsistencies in bot usernames or command syntaxes, which highlights the limitations of rule-based methods for filtering out bot-related activity. Future work could explore machine learning approaches such as BoDeGHa [37] to more accurately isolate human discussions.

Nevertheless, the identified moderate correlation between merge time and the number of comments is consistent with earlier findings from a large-scale study of 141,468 PRs by Gousios et al. [38], suggesting that PRs with longer discussion threads may also take more time to merge. Still, since not all comments are relevant to the resolution work, it is hard to correlate comment counts with actual human effort.

**Lack of Ground Truth Validation.** Without definitive ground truth data, it is difficult to determine whether these metrics accurately reflect the effort involved in resolving version conflicts. For instance, in apache/beam#4653 [39], the only PR with effort data available, developers reported spending 2.3 hours on the issue via the Jira tracking system, while the corresponding merge time was 10.1 hours and the detection to resolution time was 44.5 hours.

To improve the limited understanding of quantitative effort metrics, future work could incorporate qualitative methods such as developer interviews or surveys, as done by Pashchenko et al. [4] and by Huang et al. [40]. These approaches should better capture the nuances of human effort and could be used to validate quantitative results, although they involve more time and depend on developers' willingness to participate.

## 7.2 Incompatibility Risks of Version Conflicts

**SemVer Guarantees Backward, not Forward Compatibility.** Out of the 35 runtime errors caused by version conflicts in Table 3, 28 cases (80%) involved a scenario where a component required a newer version than the one actually loaded at runtime. In such cases, SemVer offers no protection, as it only ensures backward-compatibility, not forward-compatibility. As a result, version conflicts in which features from newer versions are expected but missing at runtime are potentially more likely to lead to software breakages.

**Reliance on Version Numbers Can Be Misleading.** Although 87.1% of the version conflicts in our dataset involve library versions that are correctly formatted according to the SemVer specification (Table 2), this alone does not imply that libraries fully adhere to SemVer's compatibility guarantees [21]. Our findings challenge the assumption that conflicts involving minor or patch version differences are always harmless. This is supported by the three concrete cases (#28, #30 and #32) in Table 3, where loading a newer patch or minor version caused runtime errors due to missing methods, thus violating SemVer's backward-compatibility rules. For example, in case #30, a deprecated method in the `jackson-databind` library was removed in the minor version range between `2.12.2` and `2.16.2`, as part of a clean-up commit in an intermediate release [41]. However, according to the SemVer specification, such removals should only occur in major releases [18]:

> *Before you completely remove the [deprecated] functionality in a new major release there should be at least one minor release that contains the deprecation so that users can smoothly transition to the new API.*

These findings also highlight the risks associated with using loose dependency constraints, such as version ranges. While such constraints can reduce the likelihood of conflicts by accepting a broader set of library versions, they do not eliminate incompatibility risks. In particular, selecting the newest version within seemingly compatible SemVer ranges may still result in failures if that version unexpectedly introduces breaking changes. As Raemaekers at al. [21] argue, determining whether a library update is backward-compatible is an undecidable problem and relying solely on version numbers is often not enough to ensure compatibility.

**Mitigating Risks of Version Conflicts.** To reduce incompatibility risks arising from version conflicts, developers should implement additional testing or validation methods to ensure that their projects continue to function correctly when library versions change. This is especially important when applying resolution strategies such as library harmonization explored in *RQ3*, where the selected version must still support all features referenced in the project. At the same time, library maintainers should closely adhere to SemVer guidelines when publishing new releases and provide clear documentation of changes to help users understand how the changes might impact their projects.

### 7.3 Version Conflict Resolution and Prevention

**Library Harmonization as the Preferred Resolution Strategy.** The analysis of resolution strategies revealed that library harmonization is the dominant approach used to resolve version conflicts in Maven-based Java projects. Harmonization of versions is reflected by strategies in *Category I* and *Category II* in our classification (Figure 6), accounting for 84 PRs (67.7% of the sample) that applied a strategy from one or both categories. This suggests a strong developer preference for either quick local fixes (e.g., adding or modifying direct dependencies) or more centralized control (e.g., alignment via `dependencyManagement` or BOMs). While local fixes may offer lower complexity, centralized approaches are generally better suited for long-term maintainability and are particularly useful in multi-module Maven projects [42].

These findings slightly differ from earlier work by Wang et al. [6], who reported that harmonizing library versions resolved 18 out of 39 conflicts in library versions (46.16%). This difference may reflect a shift in developers' preferred resolution strategies, although it could also stem from variations in manual inspection and classification methods.

**Implications for Conflict Resolution Tools.** The observed strong preference for harmonization is consistent with findings from a developer survey by Huang et al. [40], who identified harmonization as a commonly used approach for resolving version inconsistencies and developed `LibHarmo`, a tool to assist developers in applying this strategy. The tool provides version harmonization recommendations for multi-module Maven projects, while also considering the effort required to apply these changes. Our findings reinforce the value of such tools and highlight the importance of prioritizing harmonization-based strategies, with a focus on centralized solutions to preserve consistency across multiple components.

**The Potential of Conflict Prevention Strategies.** Our observations suggest the potential of proactive strategies to shift how version conflicts are typically addressed in Maven projects. While explicitly introduced in only 5 PRs, we observed preventive strategies used as conflict detection methods in more PRs. The Maven Enforcer Plugin [16] provides several built-in rules that impose constraints on dependency configuration to prevent conflicts. In particular, the `dependencyConvergence` rule enables early detection of version conflicts by failing the build at compile time when different library versions are present, thus avoiding errors that might otherwise occur at runtime. Future work could explore the adoption and long-term impact of such preventive techniques, analyzing whether they reduce the likelihood of future issues caused by version conflicts.

### 7.4 Threats to Validity

**Internal Validity.** Several steps in our methodology involved manual analysis, including the validation of identified PRs (Subsection 3.4) and the extended review of PRs to identify resolution strategies (Subsection 6.2). This introduces a potential risk of selection bias, which may lead to inconsistent interpretations or missed edge cases. To reduce this threat, we documented our procedures extensively in the paper, applied a consistent analysis approach across all samples and maintained detailed records of our decision-making process. However, a key limitation is that, due to the structure of the course, the study was conducted by a single researcher. As a result, cross-checking with additional reviewers was not possible, which may affect the reliability of our interpretations.

**External Validity.** While investigating PRs, we observed that version conflicts can lead to a broader range of runtime errors beyond those captured by our initial keyword search in Subsection 3.3. In particular, additional errors such as `NoClassDefFoundError`, `NoSuchFieldError` and `LinkageError` were sometimes associated with dependency version mismatches. Although including these terms in the keyword search phase may have uncovered additional relevant cases, we limited our search scope due to the time constraints of the study and the labour-intensive nature of the manual review phases. Therefore, our dataset may not fully capture the diversity of failures caused by version conflicts.

## 8 Responsible Research

In this section, we discuss the reproducibility and integrity considerations of our research, including data availability, compliance with open-source licensing and the use of generative AI.

### 8.1 Research Reproducibility

To support the reproducibility of our study and improve the generalizability of our findings, we described each analysis as thoroughly as possible. The data used in this study was collected from open-source repositories via the GitHub API [26]. While such data is publicly accessible, it is subject to change or removal over time. To ensure traceability, we recorded the snapshot date of our dataset collection (Subsection 3.2) and, when referencing individual PRs, we included the visited date in the corresponding citations.

To further enhance reproducibility, we provide all Python scripts used for data collection and analysis, along with the supporting datasets in both CSV and JSON formats. These resources are publicly available on GitHub [10] and include instructions for replicating the procedures.

### 8.2 Research Integrity

**Licensing of Open-Source Tools.** All open-source tools used in this study were properly cited and their licenses were respected in accordance with applicable licensing policies and academic standards.

**Use of AI.** Generative AI tools, specifically ChatGPT, were used to support the writing process of this paper, develop Python scripts, assist with data interpretation and correct LaTeX errors. A summary of the types of prompts used is provided in Appendix A. All content produced by ChatGPT was manually reviewed and verified for accuracy and correctness before inclusion in the research. ChatGPT was not used to generate new ideas and none of its textual outputs were used verbatim in the final paper.

## 9 Conclusions

This study addressed gaps in existing research on dependency hell by analyzing 124 PRs which addressed version conflicts in 85 Maven-based Java projects. We investigated version conflicts from three angles: quantifying developer effort, evaluating the reliability of SemVer as a mitigation strategy and identifying common resolution strategies. Our findings show that activity-based metrics alone offer limited insight into developer effort due to confounding factors such as unrelated changes, external delays or automation commands, pointing to the need for qualitative validation in future work.

In a focused analysis of 35 runtime errors caused by version conflicts, 80% resulted from forward incompatibilities, cases not covered by SemVer's guarantees. Moreover, despite the frequent use of SemVer syntax, three concrete cases showed that even seemingly compatible versions can break backward-compatibility, indicating the risks of relying solely on version numbers. To mitigate such risks, developers should validate selected versions through compatibility testing. Our findings also revealed library harmonization as the dominant resolution strategy (67.7% of PRs), supporting the development of version harmonization tools, especially in multi-module Maven projects. Finally, while less frequent in our dataset, proactive conflict prevention strategies, such as the Maven Enforcer Plugin, offer promising directions for solving version conflicts before they manifest at runtime.

## A LLM Prompts Used

- "What's a better word than $\langle word \rangle$ in $\langle phrase \rangle$?"
- "What's a good topic sentence for this $\langle paragraph \rangle$ discussing $\langle context \rangle$?"
- "Can you help me rephrase this $\langle paragraph \rangle$ about $\langle context \rangle$?"
- "Please format this $\langle table \rangle$ in LaTeX."
- "Is my figure $\langle caption \rangle$ logical? How can I improve it?"
- "Can you write a Python script that does $\langle task \rangle$?"
- "How can I visualize this $\langle data \rangle$?"
- "How can I fix this Python/Maven/LaTeX $\langle error \rangle$?"

## References


[1] R. Kikas, G. Gousios *et al.*, "Structure and Evolution of Package Dependency Networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 102–112. DOI: 10.1109/MSR.2017.55.

[2] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: A review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, Oct. 2007, ISSN: 1573-7616. DOI: 10.1007/s10664-007-9040-x.

[3] A. Decan, T. Mens and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, Feb. 2019, ISSN: 1573-7616. DOI: 10.1007/s10664-017-9589-y.

[4] I. Pashchenko, D.-L. Vu and F. Massacci, "A Qualitative Study of Dependency Management and Its Security Implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1513–1531, ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3417232.

[5] A. Stuckenholz, "Component evolution and versioning state of the art," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 1, p. 7, Jan. 2005, ISSN: 0163-5948. DOI: 10.1145/1039174.1039197.

[6] Y. Wang, M. Wen *et al.*, "Do the dependency conflicts in my project matter?" In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 319–330, ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236056.

[7] Y. Wang, M. Wen *et al.*, "Watchman: Monitoring dependency conflicts for Python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 125–135, ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380426.

[8] Y. Wang, R. Wu *et al.*, "Will Dependency Conflicts Affect My Program's Semantics?" *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2295–2316, Jul. 2022, ISSN: 1939-3520. DOI: 10.1109/TSE.2021.3057767.

[9] G. Robles, A. Capiluppi *et al.*, "Development effort estimation in free/open source software from activity in version control systems," *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–37, Nov. 2022, ISSN: 1573-7616. DOI: 10.1007/s10664-022-10166-x.

[10] *VladM13/maven-repo-miner*. [Online]. Available: https://github.com/VladM13/maven-repo-miner (visited on 21/06/2025).

[11] *Apache/pulsar*. [Online]. Available: https://github.com/apache/pulsar (visited on 21/06/2025).

[12] *[branch-2.7] Upgrade the BookKeeper version to 4.12.1 by hangc0276 · Pull Request #16775 · apache/pulsar*. [Online]. Available: https://github.com/apache/pulsar/pull/16775 (visited on 17/06/2025).

[13] *Spring Framework*. [Online]. Available: https://spring.io/projects/spring-framework (visited on 21/06/2025).

[14] *Introduction to the Dependency Mechanism*. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html (visited on 13/05/2025).

[15] *Apache Maven Dependency Plugin – Introduction*. [Online]. Available: https://maven.apache.org/plugins/maven-dependency-plugin/ (visited on 21/06/2025).

[16] *Apache Maven Dependency Plugin – dependency:tree*. [Online]. Available: https://maven.apache.org/plugins/maven-dependency-plugin/tree-mojo.html (visited on 06/05/2025).

[17] *Support new BungeeCord API by GatitoUwU · Pull Request #75 · lucko/BungeeGuard*. [Online]. Available: https://github.com/lucko/BungeeGuard/pull/75 (visited on 17/06/2025).

[18] T. Preston-Werner, *Semantic Versioning 2.0.0*. [Online]. Available: https://semver.org/ (visited on 20/06/2025).

[19] *Naming conventions of Maven coordinates*. [Online]. Available: https://maven.apache.org/guides/mini/guide-naming-conventions.html (visited on 29/05/2025).

[20] C. Soto-Valero, A. Benelallam *et al.*, "The Emergence of Software Diversity in Maven Central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019, pp. 333–343. DOI: 10.1109/MSR.2019.00059.

[21] S. Raemaekers, A. van Deursen and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, Jul. 2017, ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.04.008.

[22] GitHub Staff, *Octoverse 2024 Report*, Oct. 2024. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-2024/ (visited on 16/05/2025).

[23] *Maven Repository: Repositories*. [Online]. Available: https://mvnrepository.com/repos (visited on 20/06/2025).

[24] *JVM Ecosystem Report 2021*. [Online]. Available: https://snyk.io/reports/jvm-ecosystem-report-2021/ (visited on 20/06/2025).

[25] O. Dabic, E. Aghajani and G. Bavota, *Sampling Projects in GitHub for MSR Studies*, Mar. 2021. DOI: 10.48550/arXiv.2103.04682.

[26] *GitHub REST API documentation*. [Online]. Available: https://docs-internal.github.com/en/rest?apiVersion=2022-11-28 (visited on 21/06/2025).

[27] M. Griffiths-Prasolova, *The Art (and Science) of Reviewable PRs*, Nov. 2023. [Online]. Available: https://engineering.joinknack.com/art-and-science-of-reviewable-prs/ (visited on 27/05/2025).

[28] *Semver: Python helper for Semantic Versioning*. [Online]. Available: https://github.com/python-semver/python-semver (visited on 20/06/2025).

[29] *Fix[ENTESB-12609]: Spring major version conflicts by christophd · Pull Request #39 · citrusframework/yaks*. [Online]. Available: https://github.com/citrusframework/yaks/pull/39 (visited on 17/06/2025).

[30] *Build: Switch to using junit 5 bom by BenWhitehead · Pull Request #2680 · googleapis/java-storage*. [Online]. Available: https://github.com/googleapis/java-storage/pull/2680 (visited on 17/06/2025).

[31] *JUnit 5 User Guide*. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/#running-tests-build-maven-bom (visited on 18/06/2025).

[32] *Optional Dependencies and Dependency Exclusions – Maven*. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html (visited on 17/06/2025).

[33] *Re #1506 Enabling Maven (version) enforcer plugin in 'LangChain4j :: Integration :: OpenAI' module. by PrimosK · Pull Request #1508 · langchain4j/langchain4j*. [Online]. Available: https://github.com/langchain4j/langchain4j/pull/1508 (visited on 17/06/2025).

[34] *Remove use of unshaded guava (fix #1229) by tatu-at-datastax · Pull Request #1253 · stargate/stargate*. [Online]. Available: https://github.com/stargate/stargate/pull/1253 (visited on 17/06/2025).

[35] *[IOTDB-5772] spark-iotdb-connnector: Support scala 2.11 & 2.12 and resolve the Thrift version conflict with Spark's by xuanronaldo · Pull Request #9788 · apache/iotdb*. [Online]. Available: https://github.com/apache/iotdb/pull/9788 (visited on 17/06/2025).

[36] *Contrib dependency management by rakow · Pull Request #1291 · matsim-org/matsim-libs*. [Online]. Available: https://github.com/matsim-org/matsim-libs/pull/1291 (visited on 16/06/2025).

[37] M. Golzadeh, A. Decan *et al.*, "A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments," *Journal of Systems and Software*, vol. 175, p. 110 911, May 2021, ISSN: 01641212. DOI: 10.1016/j.jss.2021.110911.

[38] G. Gousios, M. Pinzger and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, New York, NY, USA: Association for Computing Machinery, May 2014, pp. 345–355, ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568260.

[39] *[BEAM-3668] Quick workaround fix for netty conflict waiting better fix by BEAM-3519 by jbonofre · Pull Request #4653 · apache/beam*. [Online]. Available: https://github.com/apache/beam/pull/4653 (visited on 16/06/2025).

[40] K. Huang, B. Chen *et al.*, "Interactive, effort-aware library version harmonization," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, New York, NY, USA: Association for Computing Ma-

chinery, Nov. 2020, pp. 518–529, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409689.

[41] *EnumResolver clean up: Remove deprecated constructors, sort · FasterXML/jackson-databind@9ea61eb*. [Online]. Available: https://github.com/FasterXML/jackson-databind/commit/9ea61eb820b82aa286ab75b56dc19ccd919a6743 (visited on 13/06/2025).

[42] B. Demers, *Maven dependency hell: Five tips to get out*, Jul. 2024. [Online]. Available: https://gradle.com/blog/five-ways-dependency-hell-maven/ (visited on 18/06/2025).