



An extended study of syntactic breaking changes in the wild

Dhanushka Jayasuriya¹ · Samuel Ou¹ · Saakshi Hegde¹ · Valerio Terragni¹ · Jens Dietrich² · Kelly Blincoe¹

Accepted: 1 October 2024

© The Author(s) 2024, corrected publication 2025

Abstract

Libraries assist in accelerating the development of software applications by providing reusable functionalities. Libraries and applications that declare these libraries as dependencies become their clients. However, as libraries evolve, maintaining the dependencies in client projects can be challenging if the new version contains breaking changes. Yet, limited research focuses on analyzing the impact of breaking changes on client projects when updating dependencies in the wild. Hence, we conduct an empirical analysis using Java projects built using Maven to investigate the impact of breaking changes introduced between two library versions. Our dataset included 18,415 Maven artifacts, declaring 142,355 direct dependencies, out of which 71.60% were not up-to-date. We automatically updated these dependencies and discovered that 11.58% of the dependency updates resulted in breaking changes that affected the client, and almost half of them were introduced during a non-major update. We analyzed the changes in the libraries that contributed towards these breaking changes, and our results indicate that changes in transitive dependencies were a significant factor in introducing breaking changes. We further investigated if it was common for clients to use functionalities of transitive dependencies directly without declaring them. This showed that over half of the clients use transitive functionality. Therefore, we analyzed actions suggested to resolve these breaking changes introduced by transitive dependencies under the discussions on open-source platforms, and the frequently suggested action was to exclude the transitive dependency from the project configuration.

Keywords Software libraries · Software dependency · Breaking changes · Software evolution · Transitive dependencies

1 Introduction

Libraries play an essential role in enhancing the efficiency and cost-effectiveness of client projects, offering reusable source code that is reliable (Mohagheghi et al. 2004; Møller et al. 2020; Foo et al. 2018; Cox et al. 2015). As with any software, these libraries evolve and release new versions, which comprise new features, improvements for existing features, and resolving existing issues (Pashchenko et al. 2018). Hence, it is vital for client projects to

Communicated by: Gabriele Bavota

Extended author information available on the last page of the article

keep their dependencies updated to benefit from the library while maintaining the security and overall health of the project (Cox et al. 2015; Pashchenko et al. 2018; Stringer et al. 2020).

However, knowing when the dependency should be updated is a challenge, as the latest versions of the dependencies can contain backward incompatible changes known as Breaking Changes (BCs) that impact clients, causing software failures. For example, renaming a method the client uses will cause a BC. These BCs can be categorized as source, binary, and behavioral BCs (Dietrich et al. 2014, 2016). Source BCs result in compilation errors, binary BCs result in linkage errors, and behavioral BCs cause the application to behave differently than expected.

Semantic versioning (Preston-Werner n.d.) is the most common approach followed by library developers to inform dependent clients whether a new version is backward compatible with the previous version (Zhang et al. 2022; Dietrich et al. 2019; Raemaekers et al. 2017; Ochoa et al. 2022b). This versioning scheme uses a three-digit number format of `Major.Minor.Patch`, and requires the `Minor` and `Patch` releases to be backward compatible. However, research has found that libraries often violate the semantic versioning scheme and introduce BCs under `Minor` or `Patch` releases (Foo et al. 2018; Raemaekers et al. 2017; Ochoa et al. 2022b).

Previous studies have focused on analyzing BCs introduced between library versions (Dig and Johnson 2006; Raemaekers et al. 2017; Koçi et al. 2019; Møller and Torp 2019). Nevertheless, it is equally important to identify the impact these BCs have on clients as not all BC functionality is used in client code. Raemaekers et al. (2017) and Ochoa et al. (2022b) analyzed the impact of BCs on client projects, but their studies were limited to only binary BCs. Furthermore, these studies concentrated on analyzing the impact of BCs introduced between two adjacent library versions, which potentially underestimate the actual impact on real-world clients who often lag more than one release behind (Kula et al. 2018; Wang et al. 2020; Salza et al. 2018) and as Stringer et al. (2020) concluded clients will not select the outdated adjacent version but will update to the currently available version. Therefore, there is a need for a comprehensive examination of the impact of BCs on clients, including both source and binary, that extends beyond the adjacent versions.

In this study, we investigate the impact of both source and binary BCs on client projects while also considering updating the dependencies “*in the wild*” by updating the client dependency to the latest stable version and examining the impact of BCs clients would encounter. To accomplish this, we conduct a large-scale empirical analysis of Java projects built using MAVEN. We analyzed 18,415 MAVEN artifacts that declared 142,355 dependencies associated with 7,454 MAVEN libraries. After updating the dependencies and examining the impact and reasons for BCs, we detected that changes in transitive dependencies contributed significantly to introducing BCs. This motivated us to further analyze the usage of transitive dependencies in clients and the actions taken to resolve BCs raised due to these transitive dependencies. Our analysis focused on the following research questions.

RQ1: To what degree are the dependencies in open-source repositories up-to-date?

We analyzed how many clients kept their dependencies up-to-date and how many clients did not maintain their dependencies. Furthermore, for the outdated dependencies, we categorized if they required a `Major`, `Minor`, or `Patch` version change to be updated to.

RQ2: How often do client-impacting BCs occur in the wild? For the outdated dependencies detected, we analyzed if updating the dependency to the latest version would cause any BCs in the client project and calculated the percentage of impact. Since BCs could exist before the latest version, we investigated the exact library version in which the client-impacting BC was initially introduced.

RQ3: What are the common types of client-impacting source BCs? Using a sample of client-impacting BCs, we conducted an in-depth analysis to understand the common types of BC changes in libraries that impact client projects

RQ4: Are client-impacting source BCs introduced in non-Major library releases? We categorized the library versions that introduced client-impacting BCs according to the semantic versioning levels. Additionally, we detect the most common change that leads to client-impacting BCs under each semantic level.

RQ5: How often do clients directly rely on transitive dependency functionality? We extracted all external functions used in the client's source code. For these external functions, we mapped them with dependencies, both direct and transitive, to the client to identify how many of them came from transitive dependencies. Furthermore, we examined whether the transitive dependency belonged to the same multi-module project as a direct dependency. If so, they are usually updated together and would not cause BCs to clients when updated. If it was a completely independent dependency, it could contribute to BCs during dependency updates.

RQ6: Can compilation error logs alone assist in determining if the BC was related to a transitive dependency? Using the compilation error logs retrieved and the functionalities of transitive dependencies used by the client extracted in prior steps, we examined if BCs related to transitive dependencies could be identified.

RQ7: How do projects currently resolve BCs caused by transitive dependencies? We looked at discussions related to transitive dependencies under open-source repositories to identify the steps software developers take to handle, avoid, and resolve BCs related to transitive dependencies.

The contributions of our study focus on analyzing the impact of source and binary BCs on client projects. Furthermore, our findings present how software developers in the community address the BCs raised due to the most frequent client-impacting syntactic BC raised during dependency updates. The key findings of our research are as follows:

- 1) 71.60% of open-source projects' dependencies are not up-to-date, meaning these clients might not fully utilize the libraries' functionalities and might contain vulnerabilities.
- 2) 11.58% of clients would encounter BCs while updating dependencies to the latest stable version; hence, updating dependencies in clients is not straightforward.
- 3) Changes in transitive dependencies are a leading cause contributing to 20.36% of client-impacting BCs. Therefore, client developers should be vigilant about using functionalities of transitive dependencies without declaring them.
- 4) Almost half of the BCs (41.58%) were observed during a non-Major dependency update, which highlights that library developers should be cautious when introducing changes during library releases.
- 5) Over half of the open-source clients (61.24%) use transitive dependencies in their projects, and most of these dependencies (64.79%) did not originate from the same multi-module project under which a direct dependency was released. Therefore, most clients using transitive dependencies may encounter issues when updating dependencies, as the transitive dependencies the clients use could be updated separately based on the requirements of the direct dependency.
- 6) Most BCs related to transitive dependencies can not be detected using only the compilation error logs since they do not accurately reflect code details contributing to the BC to trace the error location. Therefore, static analysis tools must be improved to capture the BCs encountered due to transitive dependencies, as we cannot rely on compilation error logs to assist in this process.

- 7) Software developers in open-source projects have discussed different maintenance strategies for transitive dependencies. The most frequently applied action to resolve BCs is excluding transitive dependencies, which will reduce the total dependencies added to the classpath and will prevent clients from using the transitive dependency functionality. The other frequent actions are defining the transitive dependency as a direct dependency, which will provide the correct version required for the client, and updating the direct dependency, which exposes the transitive dependency to the client project. This will bring in a new version of the transitive dependency that is compatible to the client.

Our findings provide valuable insights for client-impacting BCs. This paper significantly extends our previous conference paper (Jayasuriya et al. 2023) by further investigating the usage of transitive dependency functionality in clients without declaring them under their project (RQ5). Then, analyzing if BCs related to transitive dependencies could be detected by only using the compilation error logs (RQ6). Finally, understanding the discussions the software development community has around transitive dependencies and how to resolve BCs raised due to the clients directly using features of these dependencies (RQ7). We addressed these additional questions by analysing the data we gathered for the prior research and conducting a qualitative analysis on discussions extracted about transitive dependencies from open-source repositories. To address these additional questions, we modified Section 3, which includes both the design and the results retrieved for these questions, and Section 4, which includes the new implications drawn from our study.

The rest of the paper is structured as follows: Section 2 provides the background of our study. The design of the study and results are included under Section 3. Section 4 includes a summary of the findings, implications, and threats to validity. The related research that inspired our study is discussed in Section 5, and finally, Section 6 concludes our work.

2 Background

This section provides a background for the work conducted under this research on source and binary Breaking Changes(BCs).

Libraries are collections of reusable code exposed as APIs intended to be used by client projects. As libraries rapidly evolve and release new versions, they could add or modify existing features, fix existing issues, or enhance performance and security. A client (can be either an application or a library itself) will use these libraries as a **dependency** under the project, which will include the library along with the version or range of versions upon which the client code depends. Libraries can also depend on other libraries; therefore, when a client uses a library, it can be exposed to two types of dependencies: direct and transitive. **Direct dependencies** are explicitly defined under the project configuration and are required for the build and execution of the client project as it will directly invoke the library APIs. **Transitive dependencies**, also known as indirect dependencies, are not declared under the project configuration but are necessary for the build and execution of direct dependencies (Kikas et al. 2017). Given that transitive dependencies can themselves have direct dependencies, this creates a dependency tree, resulting in multiple levels of transitive dependencies connected with a client project. Shown in Fig. 1 is a dependency tree generated using the mvn dependency:tree command for the client 'org.opennms.newts:newts-metrics-reporter'. According to the dependency tree,

```
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli) @ newts-metrics-reporter ---
[INFO] org.opennms.newts:newts-metrics-reporter:jar:2.0.1-SNAPSHOT
[INFO] +- org.opennms.newts:newts-api:jar:2.0.1-SNAPSHOT:compile -----> Direct Dep.
[INFO] | +- com.google.guava:guava:jar:23.0:compile -----> Transitive Dep. Level 1
[INFO] | | +- com.google.code.findbugs:jsr305:jar:1.3.9:compile
[INFO] | | +- com.google.errorprone:error_prone_annotations:jar:2.0.18:compile -----> Transitive Dep. Level 2
[INFO] | | +- com.google.j2objc:j2objc-annotations:jar:1.1:compile
[INFO] | | \- org.codehaus.mojo:animal-sniffer-annotations:jar:1.14:compile
[INFO] | +- com.google.inject:guice:jar:4.0:compile -----> Transitive Dep. Level 1
[INFO] | | +- javax.inject:javax.inject:jar:1:compile -----> Transitive Dep. Level 2
[INFO] | | \- aopalliance:aopalliance:jar:1.0:compile
[INFO] | +- org.slf4j:slf4j-api:jar:1.7.12:compile
[INFO] | +- org.apache.commons:commons-jexl3:jar:3.1.1:compile -----> Transitive Dep. Level 1
[INFO] | \- org.slf4j:jcl-over-slf4j:jar:1.7.12:runtime
[INFO] \- io.dropwizard.metrics:metrics-core:jar:3.1.1:compile -----> Direct Dep
```

Fig. 1 Dependency Tree of 'org.opennms.newts:newts-metrics-reporter' client

this client has two direct dependencies, five transitive dependencies at level one, and an additional six transitive dependencies at level two. One of the direct dependencies of the client is 'org.opennms.newts:newts-api' version 2.0.1-SNAPSHOT, which has its own direct dependency on 'com.google.guava:guava' version 23.0. That makes 'com.google.guava:guava' version 23.0 a transitive dependency for the client.

Since the transitive dependency functionality is accessible to the clients, some clients use it directly in their code without declaring the dependency under the project configuration. Figure 2, shows a code snippet from the 'org.opennms.newts:newts-metrics-reporter' client. This snippet invokes the 'com.google.common.collect.Lists::newArrayList()' method defined in 'com.google.guava:guava' version 23.0. This illustrates the use of transitive dependency functionality within a client's source code.

As libraries evolve, they could often introduce unexpected behavior at either compiler, build, link, or runtime, known as incompatible changes for the clients using them compared to their previous version. These changes are known as **Breaking Changes (BCs)**. However, a BC will impact a client only if the client uses the functionality that introduced the incompatibility. BCs are broadly categorized into syntactic and behavioral (semantic) BCs. Behavioral BCs introduce changes to the behavior of the client code when executed and can only be detected

```
public void report(SortedMap<String, Gauge> gauges, SortedMap<String, Counter> counters,
                  SortedMap<String, Histogram> histograms, SortedMap<String, Meter> meters,
                  SortedMap<String, Timer> timers) {
    Timestamp timestamp = Timestamp.fromEpochMillis(clock.getTime());

    List<Sample> samples = Lists.newArrayList(); -----> com.google.common.collect.Lists.newArrayList()

    for (Map.Entry<String, Gauge> entry : gauges.entrySet()) {
        reportGauge(samples, timestamp, entry.getKey(), entry.getValue());
    }

    for (Map.Entry<String, Counter> entry : counters.entrySet()) {
        reportCounter(samples, timestamp, entry.getKey(), entry.getValue());
    }
}
```

Fig. 2 'org.opennms.newts:newts-metrics-reporter' client using functionality of 'com.google.guava:guava' by invoking com.google.common.collect.Lists.newArrayList() method

50	-	public String transform() {
58	+	public String transform() throws ScanException {
51	59	StringBuilder stringBuilder = new StringBuilder();

Fig. 3 Binary compatible but Source Breaking Change `ch.qos.logback:logback-core` when updating from version 1.1.0 to 1.1.1. (https://github.com/qos-ch/logback/compare/v_1.1.0...v_1.1.1)

at runtime and, therefore, require thorough testing (Jayasuriya et al. 2024a). Our study will only focus on syntactic BCs.

Syntactic BCs can be further divided into source and binary BCs. **Source BCs** result in incompatibilities detected during the compilation time, and **Binary BCs** result in incompatibilities manifested at the load time when linking the client application and the library binaries (Gosling et al. 2021). Both source and binary BCs stem from syntactic changes applied to the API that can be detected using static analysis tools. Some of these BCs include changes to the API signature, deletion, or renaming of a method, class, or an entire package.

Syntactic BCs often exhibit both source and binary incompatibilities, but there can be some scenarios where this could differ. For example, Fig. 3 displays a source BC in the `ch.qos.logback:logback-core` library between versions 1.1.0 and 1.1.1 by introducing a new checked exception to the method signature. The clients using this function should handle or rethrow this new exception to prevent compilation errors. This will not be detected as a binary incompatibility as the `throws` clause is not included as part of the method descriptor used in the linking process. Figure 4 displays a change in the `net.sourceforge.owlapi:owlapi-distribution` library between versions 4.3.1 and 5.0.0 which specializes the return type of a method. This strengthens the precondition and, therefore, will be source compatible, where the method used to return a `Collection` type would now return a `List`, a subtype of a `Collection`. However, it will be binary incompatible since the return type is part of the method descriptor, which is used to identify the method during the linkage process. Since the matching method descriptor will not be available during the linkage process, the clients who compiled the code with the prior library version will throw a `NoSuchMethodError` when being executed with the new version without being recompiled at first (Oracle n.d.).

We used Java projects built using **APACHE MAVEN** (Foundation 2023) as their build automation tool for the research. The **Project Object Model (POM)** defined using an XML file is the fundamental unit in **MAVEN**, responsible for the project's metadata, dependencies, and additional configurations. One of its core functionalities is dependency management, which will automatically download the dependencies along with their required dependencies, which will be transitive to the client. The most commonly used remote repository to

95	-	@NonNull
96	-	public Collection<Clause> getClauses(String tag) {
97	-	Collection<Clause> cls = new ArrayList<>();
119	+	public List<Clause> getClauses(@Nullable String tag) {
120	+	List<Clause> cls = new ArrayList<>();

Fig. 4 Source compatible but Binary Breaking Change under `net.sourceforge.owlapi:owlapi-distribution` when updating from version 4.3.1 to 5.0.0. (<https://github.com/owlcs/owlapi/compare/owlapi-parent-4.3.1...owlapi-parent-5.0.0>)

Fig. 5 Group of multi-module projects sharing the same groupId



download these MAVEN dependencies is the MAVEN CENTRAL¹, which maintains millions of software dependencies. When deployed using MAVEN, a project is called a MAVEN artifact. A MAVEN artifact is uniquely identified by its GAV coordinates, including the groupId (G), representing the organization or group responsible for the dependency, the artifactId (A) signifying the identifier for the artifact within the group, and the version (V) denoting the artifact's version. If a project contains multiple modules, it will adopt a parent-child structure, where a parent will contain multiple child artifacts. In Maven, the multi-module projects usually follow the pattern of sharing the same groupId or appending an identifier to the parent's groupId² and deferring from the artifactId. Figure 5 shows how the 'OW2 ASM' libraries share the the same groupId and they have different artifacts based on the modules. All these 'OW2 ASM' libraries belong to the same multi-module project.

These multi-module projects usually depend on one another and, therefore, are built and released together. Clients who use these multi-moduled dependencies usually define a variable in the build script to consistently refer to the version of all artifacts within the same multi-moduled project.

```
<dependency>
  <groupId>org.yaml</groupId>
  <artifactId>snakeyaml</artifactId>
  <scope>compile</scope>
  <version>1.30</version>
</dependency>
```

Listing 1 A Dependency Block in a pom.xml file

When these artifacts are used as dependencies in client projects, additionally to these coordinates, the scope, which indicates at what life-cycle phase (build, runtime, test) the dependency will be added to the client's project classpath, can be included. The dependency will be declared under the <dependencies> section within the client's pom.xml file, which is illustrated under Listing 1.

3 Study Design and Results

This section discusses the study's design to collect, process, and analyze the data. Illustrated in Fig. 6 is the overview of the process we followed for the study. We first collected repositories containing MAVEN artifacts and analyzed their dependency usage to detect if they maintain up-to-date dependency versions. Next, we updated the outdated dependencies one at a time and recompiled the artifacts. The artifacts that failed to compile provided us with the total client projects that were impacted due to BCs during dependency updates. We extracted the compilation error for each dependency update that failed to compile, then mapped it with the change in the library between the two versions that caused the BC to identify the common

¹ <https://search.maven.org/>

² <https://maven.apache.org/guides/mini/guide-naming-conventions.html>

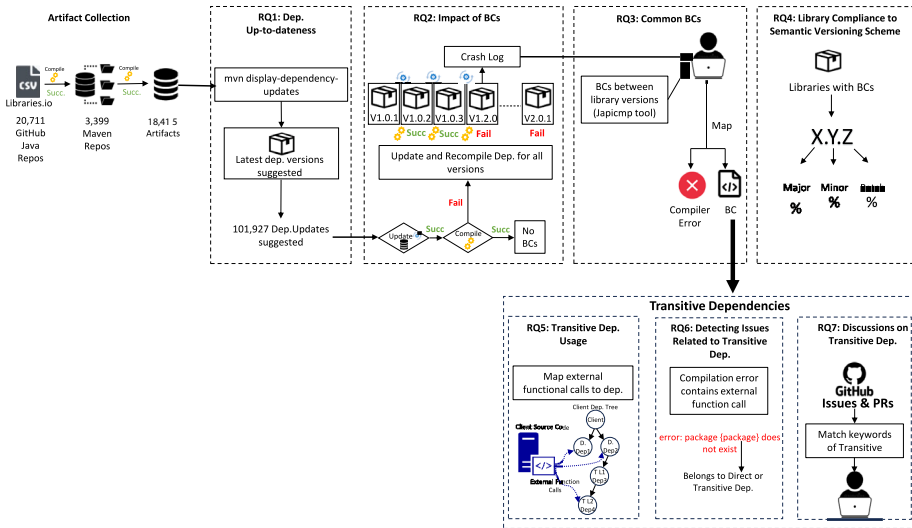


Fig. 6 Overview of the Study

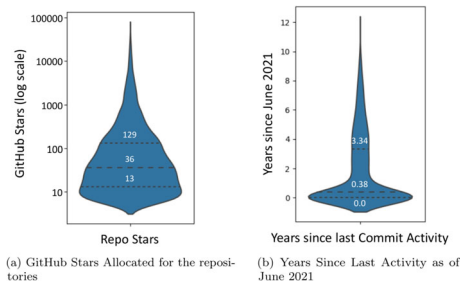
changes that impact client projects. We used the JAPICMP tool to extract the client-impacting syntactic BCs. We compared it with the results from compiling the artifacts after updating dependencies to investigate if static analysis tools are reliable sources in detecting BCs. We analyzed if libraries comply with the semantic versioning scheme by analyzing the BCs introduced during dependency updates at different semantic version levels. As changes in transitive dependencies were a significant factor in introducing BCs, we extended our study to identify how many clients rely on transitive dependency functionalities and if compilation errors could assist in determining if the BCs were related to transitive dependency changes. Finally, we analyzed discussions on transitive dependencies under open-source repositories available on GitHub to understand how the software engineering community uses and handles BC related to transitive dependencies under their projects. The study data, results, and scripts used to acquire and analyze the data are available in the replication package (Jayasuriya et al. 2024b)

3.1 Experiment Setup

Artifact Collection For the study, we selected Java repositories, as we had to analyze the source code of the projects, and the researchers had expertise in Java. Additionally Java is one of the widely used programming languages, and there are many static analysis tools built to detect both source and binary BCs. We collected the repositories for the analysis using the Libraries.io dataset (Inc Tidelifit 2022), aligning with similar research conducted in this area (Decan and Mens 2021; He et al. 2021; Alfadel et al. 2021). We considered these repositories as client repositories for our study. We first selected the Java repositories from the dataset and applied the following filters. Available on GitHub, as we needed to clone them to the local machine for further analysis and repositories that are not fork repositories to avoid analyzing multiple copies of the same repository (Kalliamvakou et al. 2014). Using the GitHub API³, we filtered repositories with at least five GitHub stars to ensure the quality

³ <https://docs.github.com/en/research>

Fig. 7 Repository distribution by Stars and Last Commit Activity



of the repositories following similar research in this area (He et al. 2021; Mujahid et al. 2020). We cloned the selected repositories to the local machine on June 2021, which provided us a total of 20,711 repositories. Next, we selected the repositories that have configured MAVEN as their build tool by verifying if they have configured a `pom.xml` file, which reduces eligible repositories for the study to 7,019. We only considered repositories that compiled successfully to identify BCs through new compilation errors generated while automatically updating the dependency under our analysis for RQ2. Compiling the projects was time-consuming as it required resolving all dependencies to the local repository. We maintained the default MAVEN local repository settings, prioritizing project-specific repository settings if configured in an individual project's `pom.xml` file. If local repository settings were configured for a project, it would take precedence over the project's `pom.xml` file, and MAVEN would download the dependencies based on the local repository settings.

We compiled the projects using Java 8 and reran the failing projects using Java 11. We chose these versions since, as of 2014-2021, the period in which the repositories had committed activities, they were the only Java releases that received Long-Term Support from Oracle. After compiling the repositories using the MAVEN command, we successfully compiled 3,211 and 188 repositories (3,399 repositories in total) on Java 8 and Java 11, respectively. Even though the number of projects compiled on Java 11 was few, it increased the total number of successfully built projects by 9%. Figure 7a represents the distribution of GitHub stars for the these selected projects, where the median number of stars is 36. Further, Fig. 7b shows the last commit activity of the repositories used for the analysis as of year 2022. More than half of these projects had their last commit activity within less than an year, as indicated by a median years since June 2021 being 0.38 years.

Following the same procedure used to compile the repositories, we compiled each MAVEN artifact in these repositories containing a unique GAV coordinate. There were 18,415 MAVEN artifacts that compiled successfully within the 3,399 repositories. These individual MAVEN artifacts will be the client projects we use for the analysis. Our research focuses on examining the dependencies specified in these clients and evaluating the individual impact of updating each dependency, following a similar methodology conducted under a prior study (Harrand et al. 2022).

3.2 RQ1: Dependency Up-To-Dateness

In this section, we answer **RQ1**: “To what degree are the dependencies in open-source repositories up-to-date?”

3.2.1 Method

Using MAVEN commands, we extracted 142,355 direct dependencies declared under the 18,415 MAVEN artifacts. To determine if these direct dependencies were up-to-date, we used the `display-dependency-update` MAVEN command to check if new versions were available. This command provides the latest versions for the outdated dependencies, and these suggestions could include either a `Major`, `Minor`, or `Patch` update. Some of the suggested library versions included ‘alpha’, ‘beta’, ‘SNAPSHOT’, and ‘RC’ suffixes in them, which are not stable releases according to the MAVEN Repository site and previous studies (Olivera 2022; Ochoa et al. 2022b). Therefore, we excluded these unstable versions and retrieved the latest stable versions for the libraries. For our study, we did not consider Java as a dependency for the project and its potential impacts when updating its version, as Java versions are widely recognized as being backward compatible (Darcy 2021). Hence, for this research, we maintained the Java version as a constant based on which it was successfully compiled initially. To determine the degree of outdatedness for the dependencies that are not up-to-date, we developed scripts to assess whether the suggested library version would require a `Major`, `Minor`, or `Patch` update.

3.2.2 Results

Assessing the current and latest suggested versions for each dependency, we observed that 71.60% of the dependencies in these clients were outdated. This is similar to the findings of Salza et al. (2018), who reported that 63% of the external libraries in mobile applications are never updated.

In Maven-built Java projects, dependencies are defined under different scopes, determining their inclusion in the project classpath during compilation, runtime, or testing. Table 1 presents the breakdown of the extracted dependency scopes. It further illustrates that regardless of the scope of the dependencies all other dependencies except for system scope dependencies are generally outdated. System scope dependencies are contained within the project JAR and are not retrieved from the repository, so the latest versions for system scope dependencies are not suggested.

Table 2 provides the dependency updates suggested broken down according to the semantic version level the update belongs to. For the total dependencies extracted from the clients, MAVEN proposed a `Major` update for 22.10% of the dependencies, a `Minor` update for 32.58%, and a `Patch` update for 16.02% of the dependencies. An additional 537 updates suggested could not be programmatically categorized into what semantic versioning level it

Table 1 Dependencies Analyzed with their Scopes

Dependency Scope	Number of Dependencies	Number of outdated Dependencies	Percentage of outdated dependencies
<code>compile</code>	57,280	47,326	82.62%
<code>runtime</code>	1,181	958	81.17%
<code>provided</code>	13,588	10,406	76.58%
<code>system</code>	19,629	4	0.02%
<code>import</code>	7	6	85.71%
<code>test</code>	50,670	43,227	85.31%

Table 2 Dependency Updates Suggested Based on the Semantic Versioning Level

Update Version level	Number of updates	Percentage of updates
Major	31,464	22.10%
Minor	46,376	32.58%
Patch	22,818	16.02%

belonged to because they did not adhere to the correct semantic versioning scheme. Examples of some of the unclassified version updates include 9+181-r4173-1 -> 9-dev-r4023-3, 1.r.69-SNAPSHOT -> 1.r.69.20210929, 2.0.B1 -> 2.0.M1.

Based on these results, dependency updates were suggested for 11,744 clients, indicating that 43.79% of clients had at least one outdated dependency. This complements the research conducted by Wang et al. (2020), who reported that 54.9% of projects do not update half of their dependencies. Comparing with the results 81.5% of the software projects keep their dependencies outdated, which (Kula et al. 2018) reported the maintenance of dependencies in open-source projects has increased over time.

Answering RQ1: *To what degree are the dependencies in open-source repositories up-to-date?* Our analysis indicated that 71.60% of open-source projects' dependencies were not up-to-date. Among these outdated dependencies, 22.10% had a Major update available, while Minor and Patch updates were available for 32.58% and 16.02%, outdated dependencies, respectively. Considering all the artifacts used for the analysis, 43.79% had at least one outdated dependency.

3.3 RQ2: Impact of BCs

In this section, we answer **RQ2**: "How often do client-impacting BCs occur in the wild?"

3.3.1 Method

This section identifies the BCs introduced during dependency updates on client projects. We considered all Major, Minor, and Patch updates suggested in the previous section, as prior research confirms that BCs are introduced not only in Major but also in Minor and Patch versions as well (Raemaekers et al. 2014; Brito et al. 2018a; Ochoa et al. 2022b). We first focused on detecting BCs introduced due to source incompatibilities between library versions. This process involved updating the current version of the dependency to the latest version suggested by MAVEN and verifying if the dependency update caused compilation errors. Suppose the update failed to compile; this would signify that a BC could exist in the latest version. We updated the dependencies to the latest stable version, aiming to replicate real-world dependency updating practice. In a practical scenario, when a project updates an outdated dependency, it is more likely to select the latest stable release than the next adjacent version.

We developed a script to systematically update an individual outdated dependency at a time in the `pom.xml` file and compile the client. We applied this process for each outdated dependency under each client. We could not automatically update 31,085 dependencies, constituting 30.49% of the total dependencies due to various factors such as the dependency

or its version not being available on the `pom.xml` file or the dependency version declared as a variable, which affects the entire project version and its dependencies. When updating the dependencies for the client projects defined within a parent project, known as child projects, the dependency could be declared either in the child `pom.xml` file or the parent `pom.xml` file. Therefore, depending on where the dependency was updated, the script compiles either the child project or both the parent and the child project to assess whether the dependency update resulted in compilation errors.

By analyzing the execution log, we assessed if the client compiles successfully or not after each dependency update. In instances where it was successful, it signifies that no BCs were encountered during the dependency update. Hence, we assigned it as a 'Successful'; if the build failed, we marked it as a 'Fail'. It is important to note that all build failures are not necessarily linked to source BCs introduced between library versions. There were instances where we updated a group of dependencies together since the dependency version was defined as a shared property, and if the latest version was unavailable for all the grouped dependencies, this led to a build failure. Therefore, to determine whether the build failure was related to the latest version being unavailable and to identify the exact version at which the build failure was introduced, we recompiled the failed clients for the library versions between the current and the latest.

For every dependency update that resulted in build failures, we wanted to determine the initial version that introduced this compilation error. This allowed us to identify when the BC was first introduced and under which semantic version level. For this, we collected all version numbers released by a library between the current and the suggested latest version that introduced the BC, using Maven Central (Olivera 2022) repositories. Starting from the version adjacent to the current one, we created a script to automatically update the declared dependency and recompile the project for each subsequent version until the version that introduced the BC was detected.

To validate if potential BCs could be detected using static analysis tools, we used the JAPICMP tool (version 0.16.0), which can detect syntactic BCs introduced between two library versions. We chose this tool as its GitHub repository⁴ is continuously maintained, having its last commit recorded in November 2023 (as of January 2024), and this tool has also been used in prior research (Ochoa et al. 2022a, b). This tool takes in two versions of a jar file and extracts all changes between the two versions. Then, these changes are classified as compatible or incompatible, and the incompatible changes are distinguished as either source, binary, or both source and binary incompatible. Since the JAPICMP tool needs the jar files of the library versions to detect the syntactic BCs, we developed a script to download all dependency jars from the Maven Central repositories using the previously extracted version numbers. Next, for each library, we passed the consecutive library versions into the JAPICMP tool to compute the total source and binary BCs introduced with each version update. We used these counts obtained for the source and binary BCs to determine the total BCs introduced by the versions that did not impact the clients.

To automatically detect whether the BCs reported by the JAPICMP tool impact the clients, we extracted the library functionalities used by these clients. Then, we verified if the tool reported those library functions with BCs. To extract the library functional calls a client uses at class, method, or field level, we used the ASM Framework (Bruneton et al. 2022). ASM analyzed the bytecode generated for the clients and detected all the functional calls

⁴ <https://github.com/siom79/japicmp>

under the client. The analysis at the class level included annotations, class types, and super types, which included both interfaces and superclasses. We analyzed the method signature at the method level, including exceptions thrown, annotations used, parameters, and return types with their generics. We looked at the local variables, method invocations, object instantiation, exceptions, and try-catch blocks inside a method. Next, we analyzed the fields defined at the class level, including the field types, generics used in the field type, and the annotations.

For all the functional calls extracted, we created a script to determine the external functional calls, excluding the inbuilt Java functionality. We then mapped these calls to the relevant dependencies by identifying all dependencies linked to the client using the MAVEN dependency tree command. This command provided both direct and transitive dependencies linked to the client and the level at which transitive dependencies were linked to the client. In cases where the same library existed in different versions at various levels of the dependency tree, we applied the dependency mediation (Foundation 2023) technique to map the external calls with the nearest dependency linked to the client. This approach mapped all external functional calls to the dependencies linked to the client.

We cross-referenced the external calls mapped to a library with the reports generated by the JAPICMP tool for each adjacent library version to detect if BCs were reported for those external functional calls. Our analysis only focused on external functional calls mapped to the direct dependencies, as our research only focused on the impact of BCs arising from direct dependencies.

3.3.2 Results

For all the outdated dependencies extracted in response to RQ1. We could automatically update 69.50% (70,840 out of 101,925) of these outdated dependencies, including successful and failed dependency updates. Out of the total dependencies that could be automatically updated, 56,003 of the dependency updates, which is 79.05% of the updates, were successful, while 14,837 of the updates, which represents 20.97% of the updates, failed. Then, for the failed dependency updates, we recompiled the clients for all library versions between the current and the latest version to detect the first version that introduced the BC. After recompiling the clients for all versions between the current and the latest, we identified that the version that introduced a BC was the latest suggested version for 12.58% of the dependencies and was a version prior to the latest version for 87.42%.

Analyzing the crash logs associated with these build failures, we observed that only 8,207 instances, which is 55.31% of the total build failures, contained a compilation error in the crash log. This implies that, among the total dependency updates that were successful and failed, source BCs were encountered for 11.58% of the updates. This client-impacting source BCs is higher than reported in prior research (Raemaekers et al. 2017; Ochoa et al. 2022b) which used the adjacent library version to identify the impact of binary BCs on client projects. It is noteworthy that this figure accounts for all unique dependencies; however, in some instances, dependencies that belong to the same sub-module projects (for example, a set of dependencies with a shared `groupId` that are released together and therefore must be updated together). For some instances, a shared parameter was used in the `pom.xml` file to ensure that the dependency updates were synchronized among these projects. 1,611 compilation errors were recorded as unique dependency updates that failed but could be considered duplicate counts as those dependencies belong to the same sub-module project. If these duplicate compilation errors were excluded from the total count of source BCs, the impact on client projects would

decrease to 9.31% of the dependency updates but still surpasses figures reported in prior research (Raemaekers et al. 2017; Ochoa et al. 2022b).

We further analyzed how many dependencies leading to source BCs involved the use of library internal APIs, as internal APIs do not guarantee API compatibility between versions. For the library functions used by the clients, we checked if the 'internal' keyword was included in the package structure. Our results show that 820 client projects (4.8% of the total clients) used internal APIs from 932 dependencies (0.65% of the total dependencies). When these dependencies were updated, 192 led to source BCs, accounting for 2.34% of the total source BCs. This indicates that 20.6% of updates on dependencies using internal APIs led to source BCs, implying that clients using internal library APIs have a higher risk of encountering BCs during dependency updates.

From the build failures that did not result in compilation errors, we randomly selected a subset of 30 logs and manually analyzed them, which revealed that majority of these failures had unresolved dependencies and Maven configuration issues. For the updates that reported compilation errors, we counted all the successful builds between the current and the version introducing the BCs, resulting in a build failure. Considering these successful dependency updates and the number of dependency update failures, we concluded that BCs resulted in 4.35% of the updates when updating the dependencies to the adjacent version. Notably, this aligns with findings from previous studies by Ochoa et al. (2022b) and Xavier et al. (2017), who updated dependencies to adjacent versions and reported results of 7.9% and 2.54%, respectively, as client-impacting binary BCs. However, it is essential to note that these figures might not reflect realistic scenarios, as projects are more likely to update to the latest stable version during dependency updates.

Using the library functional calls made by the client and the output from the JAPICMP tool, we verified whether the version reported by the JAPICMP tool as containing source BCs aligns with the breaking version identified through detecting compilation errors. We conducted this experiment to understand if the JAPICMP tool reports consistent results compared with the results we obtained through compiling the clients. In 18.53% of the dependency updates that resulted in compilation errors, the tool also reported source BCs. For 13.90% of instances, the JAPICMP tool reported source BCs for a library version released prior to the version in which the compilation error occurred; for 4.81% of instances, it reported source BCs for a version released after the version in which the compilation error occurred. Therefore, the results from the JAPICMP tool do not align with those obtained by identifying compilation errors. This mismatch is mainly because the tool analyzes BCs in isolation and not the context in which it is used by the client when making decisions.

Next, we extracted both source and binary BCs using the JAPICMP tool for all the dependencies that could be automatically updated. For these dependencies, the tool reported 9,221 binary BCs, which is 13.01% of the total dependency updates, and 11,444 source BCs, representing 16.15% of the total dependency updates. Notably, the number of source BCs detected by the JAPICMP tool exceeds the count of source BCs identified through compiling the clients. This could be attributed to factors such as the JAPICMP tool reporting a BC when the superclass of a class changes, even if the functionality within the new superclass remains the same as the previous superclass, which does not impact the client. Another instance is when a set of classes is removed from a library and added as a dependency to the library; this is reported as a BC since the classes were removed. However, in reality, those classes are still available for the client as a transitive dependency. Therefore, some of the BCs reported by the JAPICMP tool are false positives.

Answering RQ2: *How often do client-impacting BCs occur in the wild?* Our results indicate that, after updating a dependency in a client and compiling it, the build failed for 11.58% of the dependency updates. When the dependencies were incrementally updated to the adjacent version and considering all successful client compilations, BCs were encountered for 4.35% of the dependency updates. Although the BCs reported by the JAPICMP tool exceeded the BCs identified through the analysis of compilation errors, it is essential to note that some of the BCs reported by the JAPICMP tool are false positives because the tool does not assess the context in which the BC functionality is used in the client code and instead, considers the change in isolation when reporting BCs.

3.4 RQ3: Common BCs

In this section, we answer **RQ3**: “What are the common types of client-impacting source BCs?”

3.4.1 Method

We used client-impacting syntactic BCs identified in the previous RQ to conduct a manual analysis to detect the change in the library that caused the client to fail compilation. For this, we first extracted the compiler error messages that were generated. When extracting the messages, we omitted the code-specific information, such as the class or package name and any reference details, to help us group similar error messages together. We replaced the code-specific information with a ‘...’ notation for consistency. For example, ‘package org.osgi.util.tracker does not exist’ was converted as ‘package ... does not exist’. Following this approach, we identified 105 distinct types of compilation error messages. In certain crash logs, multiple compiler error messages were encountered, and the same compilation error message appeared in different locations within the client code. Each occurrence of a compilation error message was treated as a separate instance, as they could be linked to different changes in the library, and there were 94,737 compilation errors in total. To conduct the manual analysis, we first sampled the compilation error messages with a confidence interval of 95% and an error rate of 5% (Wonnacott and Wonnacott 1991), leading to a sample size of 380. We followed the stratified sampling strategy to create a dataset that included all compilation error messages for the analysis. For the selected sample, we created a comprehensive dataset that included the following details: the crash log, the line number within the crash log, the compilation error message, the dependency with its current and breaking versions, the output generated by the JAPICMP tool, the GitHub URL for the respective client project, the class where the compiler error occurred, and the Git-Diff Web URL link for the library versions available on GitHub. We adhered to the thematic analysis for the manual analysis, an approach designed to identify patterns in qualitative analysis (Cruzes and Dyba 2011). To label the data, we developed codes representing various types of changes in the libraries that lead to BCs. We followed an integrated approach (Cruzes and Dyba 2011) when developing the codes, which allows us to create predefined codes for the analysis and later derive codes while conducting the manual analysis. We created 101 predefined codes, categorized under Package, Interface, and Class levels based on syntactic BCs introduced between two library versions that were presented by prior researchers (Brito et al. 2018a; Dig and Johnson 2006; Xavier et al. 2017) and using an article which describes these BCs published by IBM (des Rivières 2017). Two authors, familiar with Java programming, did the manual analysis to

identify the change between the library version that caused the compilation error. Following the recommendation by O'Connor and Joffe (2020) to apply multiple coding between 10-25% of the data, we opted to use 20% of the data from the sampled dataset for the multiple coding. This resulted in a sample size of 68 records, chosen using a stratified sampling technique to ensure the inclusion of different compilation error messages in the sample set.

To familiarize ourselves with the coding and assess the coding agreement, we randomly selected four samples from the initial data sample, which were not part of the 68 samples selected for multiple coding. Each coder independently labeled the samples and then checked for agreement. Disagreement on more than one label would cause the agreement to fall below 75%, which is below acceptable (Zach 2021). For our first coding round, the agreement was below the acceptable threshold; hence, we jointly discussed how we extracted the coding values and the issues we had while coding. We repeated this process until an agreement of more than 75% was achieved, and both coders were confident in the coding process.

Then, both coders independently performed the coding for the 68 sample records. To assess the inter-rater reliability of the coded values, we used Cohen's Kappa coefficient, a statistical measure to calculate the coders' agreement for qualitative analysis (El Emam 1999). The Kappa score is calculated based on the two coders' agreement and the random agreement. We obtained a Kappa score of 0.68 for the multiple coded values, indicating substantial agreement based on the interpretation of the Kappa score value. Since there was substantial agreement and it took approximately five to ten minutes to analyze one record, only one of the authors coded the remaining samples in the manual analysis set. The results are presented in the following section.

3.4.2 Results

Table 3 displays the ten most occurring compilation error messages, with their occurrence frequencies.

During the manual analysis, we observed that five compilation errors were not reproducible, and another five types were generated under Groovy classes. We excluded these errors as we only focused on Java-related compilation errors, which reduced the number of compilation errors to 96. The labels derived from the manual analysis provided more insight into the library changes that lead to BCs impacting clients. These changes include the new

Table 3 Top Ten Compilation Errors which had the most Occurrences

Compiler Error	Occurrence	Percentage
cannot find symbol	54,194	57.20%
package ... does not exist	20,405	21.53%
method does not override or implement a method from a supertype	3,186	3.36%
incompatible types: ... cannot be converted to	2,403	2.54%
cannot access	2,333	2.46%
reference to ... is ambiguous	1,722	1.82%
static import only from classes and interfaces	1,587	1.68%
no suitable method found for	1,075	1.13%
is not abstract and does not override abstract method	890	0.94%
method ... cannot be applied to given types	871	0.92%

Table 4 Top ten changes in libraries which causes Source Incompatibility with its percentage of occurrence

Library Change	Occurrence %
Change result type of method in class	5.68%
Delete package	5.09%
Delete class	3.89%
Rename package	2.10%
Delete type parameters from class	2.10%
Decrease access of constructor in class	2.10%
Delete method from class	1.79%
Delete interface method	1.50%
Delete interface	1.50%
Delete checked exceptions thrown from method in class	1.50%

version of the library being compiled with a version of Java that is incompatible with the client, a dependency of the library being modified, incompatibilities with not updating dependent libraries together, or adding a deprecated annotation to a class, interface, or method. We encountered incompatibilities due to changes related to the type parameter (generics in Java), which was introduced in the article by IBM (des Rivières 2017) but was mentioned as a limitation for the research conducted by Ochoa et al. (2022b).

The manual analysis showed that changes in transitive dependencies were the most common factor, which led to 20.36% of the BCs. This situation arises when the client under analysis uses the functionality of the dependencies of the direct dependency. When the dependencies of the direct dependency change and those changes are not compatible with the client using them, it leads to BCs. Incompatibility with another library when updating a dependency independently and incompatibilities related to the Java version used to compile the library occurred 3.89% and 3.45% of the instances, respectively.

Table 4 shows the top ten changes in the library that led to source incompatibilities and their corresponding occurrence percentages. The results indicate that changing the result type of a method⁵ in a class is a common change across libraries which introduced source incompatibilities to clients. This change is shown under Fig. 8, which shows that the ‘getParentNode’ method, which previously returned the Node Type object, has now changed to return an Optional Type Node object, which causes BCs to clients using this API. Deleting an API package or a class were the other changes that significantly impacted clients. Figure 9 shows that the ‘org.mockito.runner’ package was removed there for the ‘MockitoJUnitRunner’ class was also removed, which led to BCs in clients. Figure 10 shows that the package ‘org.axonframework.commandhandling’ was renamed to ‘org.axonframework.commandhandling.annotation’, which again led to BCs for clients using API end points from the ‘TargetAggregateIdentifier’ class.

Our analysis showed that the same BC can impact clients differently based on how the functionality is used. For instance, the ‘Change result type of method in class’ BC will affect clients based on how the method’s return value is used. It could be used in a loop iteration, conditional statement comparison, or a method on the value returned. If the return type differs for all these scenarios, it will impact the client. Thus, understanding the context in which the dependency is used is crucial for the dependency update process. Throughout the analysis,

⁵ The term ‘changing the result type’ was taken from des Rivières (2017) and can also be referred to as ‘changing the return type’ of a method.

318	-	public Node getParentNode() {
319	-	return parentNode;
347	+	public Optional<Node> getParentNode() {
348	+	return Optional.ofNullable(parentNode);

Fig. 8 Change result type of method in class under `com.github.javaparser:javaparser-core` when updating version from 2.5.1 to 2.24.0 (<https://github.com/javaparser/javaparser/compare/javaparser-parent-2.5.1...javaparser-parent-2.24.0>)

we encountered challenges in determining the cause of specific compilation errors by only considering the possible syntactic changes. 16 of the samples (4.97%) analyzed were related to code generation libraries, and the incompatibility for these samples did not arise from a syntactic BC introduced between the two versions of the library. Through analyzing the source code changes between the library versions, we concluded that the issue was linked to the changes in the logic that is used to generate code in the client's code base during compilation. Therefore, changes in code generation libraries that lead to BCs in client projects will differ from other changes that cause syntactic BCs and, as a result, might not be detected as BCs by the static analysis tools. We could not identify the cause for the BCs for another 22 samples (6.83%), and we assume it to be associated with transitive dependency changes.

Answering RQ3: *What are the common types of client-impacting source BCs?* The results of the manual analysis indicated that changes in transitive dependencies were the most common reason for the syntactic BCs that impact clients during dependency updates. Notably, the two most common changes in the library that contributed towards source BCs were changing the result type of a method in a class and deleting an API package. The other library changes that caused syntactic BCs did not significantly contribute compared to all changes.

3.5 RQ4: Library Compliance to the Semantic Versioning Scheme

In this section, we answer **RQ4**: “Are client-impacting source BCs introduced in non-Major library releases?”

3.5.1 Method

For the library versions that introduced syntactic BCs, we analyzed whether they included non-Major releases and identified the common change under those releases that contributed towards client-impacting syntactic BCs under each semantic versioning level. Additionally, we explored how the BCs arise due to direct and transitive dependencies distributed under the semantic version levels.

74 src/org/mockito/runner/MockitoJUnitRunner.java

Fig. 9 Delete Java Package under `org.mockito:mockito-core` when updating from 1.10.19 to 4.0.0 (<https://github.com/mockito/mockito/compare/v1.10.19...v4.0.0>)

```

17 - package org.axonframework.commandhandling.annotation;
14 + package org.axonframework.commandhandling;
39 ↕
36 public @interface TargetAggregateIdentifier {
    
```

Fig. 10 Rename Java Package under org.axonframework:axon-core when updating from 2.1.2 to 3.0-M1 (<https://github.com/AxonFramework/AxonFramework/compare/axon-2.1.2...axon-3.0-M1>)

3.5.2 Results

Table 5 presents the counts for the syntactic BCs that impact clients at different semantic version levels. Specifically, 58.41% of the BCs resulted during a Major update, while 33.49% and 8.09% resulted during Minor and Patch update, respectively. Consequently, non-Major updates accounted for nearly half of the identified BCs.

Among the manually analyzed data sample, 147 instances belong to Major updates, 138 to Minor updates, and 45 to Patch updates. The most common changes in the libraries under Major updates that resulted in BCs in clients were changing the result type of a method in a class, deleting a class, renaming an API package, and deleting an API package. For Minor updates, the common BCs changes were deleting an API package, changing a static method to a non-static method in a class, and changing the result type of a class method. No BC significantly contributed to incompatibilities during Patch updates.

We analyzed how BCs introduced by direct and transitive dependencies were distributed across various semantic version levels. As depicted in Table 6, Major updates more frequently contained BCs introduced by direct dependencies than non-Major updates. Conversely, BCs introduced by transitive dependencies are notably common in non-Major updates. Even though the semantic versioning scheme defines rules when BCs are allowed during library releases, many developers will not be aware of the BCs introduced by transitive dependencies.

Answering RQ4: *Are client-impacting source BCs introduced in non-Major library releases?* Most BCs impacting clients were introduced during Major updates. However, violating the semantic versioning scheme, 41.58% of the BCs were introduced during a non-Major update. When focusing on BCs introduced due to transitive dependency changes, it was evident that non-Major updates contained most of these BCs.

3.6 RQ5: Transitive Dependency Usage

In this section, we answer RQ5: “How often do clients directly rely on transitive dependency functionality?”

Table 5 Update level in libraries and the number of impacted artifacts

Update Level	Number of Impacting Artifacts	Percentage
Major	4,742	58.41%
Minor	2,719	33.49%
Patch	657	8.09%

Table 6 Distribution of BC introduced by Direct and Transitive Dependencies at each Semantic Versioning Level

Source of BC	Semantic Version Level		
	Major	Minor	Patch
Direct Dependency	48.68%	37.72%	13.60%
Transitive Dependency	38.57%	54.28%	7.14%

3.6.1 Method

According to our analysis of the prior RQs, transitive dependency changes had a significant impact on introducing BCs to client projects. Therefore, we conducted an analysis to determine the degree to which clients directly use the functionality of transitive dependencies, as described in the discussion on transitive dependencies in Section 2. We used the methodology explained in section 3.3.1, which uses the ASM framework to extract the direct use of transitive dependency functionalities from client projects. The detected transitive functionalities provided an understanding of how often transitive dependency functionalities are used in client projects. Then, we used the `mvn dependency:tree` command to identify the level at which a transitive dependency is connected to the client project, providing us with the level to which transitive dependencies are utilized.

Further, for the collected transitive functionality that clients used, we checked how many of them were related to the same multi-module project. Using the functionality of a multi-module project should not ideally cause issues as they are guaranteed to work together by the library developers and are compatible with one another, known as blossom compatible (Dann et al. 2023). Thus, they are updated and released together to maintain compatibility. However, transitive dependencies that do not belong to the same group of projects can create issues as they will change and can be updated on the requirement of the direct dependency defining it. Therefore, we examined the `groupId`s of the transitive dependencies and their associated direct dependency to identify how many were part of the same multi-module project.

3.6.2 Results

We observed that 61.24% of all clients directly used functionalities of at least one transitive dependency connected with the project via direct dependencies. This observation aligns with the research of Kikas et al. (2017), who confirm that transitive dependency usage is widespread in software ecosystems like Javascript.

The bar chart in Fig. 11 displays the distribution of clients utilizing transitive dependency functionalities at various levels. This chart illustrates that 57.44% of clients utilized transitive dependency functionality at level one. While the usage of level two and level three transitive dependency functionalities was less prevalent than at level one, it accounted for 25.91% and 7.70% of usage, respectively.

We only extracted transitive dependencies up to level ten, and we observed that none of the clients used the functionality of transitive dependencies beyond level seven. Only two clients utilized transitive dependency functionalities at level seven, contributing to 0.01% of the total clients analyzed.

We observed that a client using a functionality of a transitive dependency, which is deeper in the dependency tree than a level one transitive dependency did not always use functionalities of transitive dependencies at prior levels. Additionally, 16.34% of clients did not use the

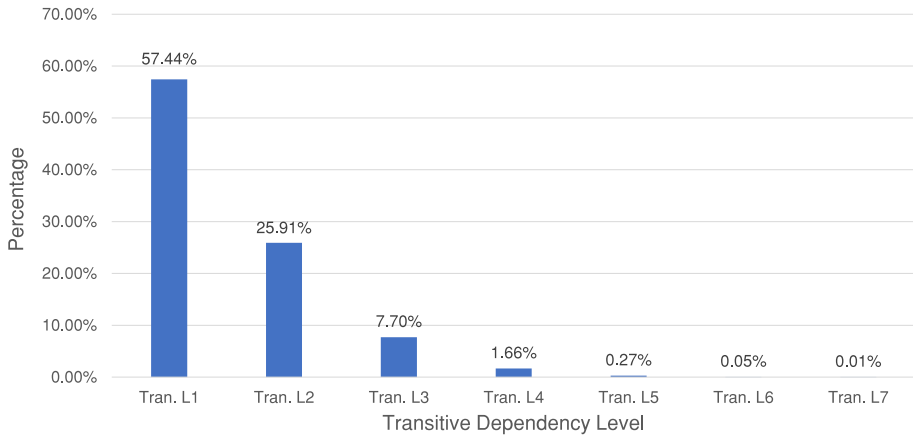


Fig. 11 Bar Chart representing maximum level of Transitive Dependency usage in client projects

direct dependencies’ functionality; from those clients, 41.58% did use the transitive dependencies’ functionality. This led us to analyze if all direct dependencies contained source code that exposed APIs for the client projects. If not, was the dependency a wrapper that aggregates all related dependencies to be included in the project so that one dependency could be defined instead of many dependencies?

An example for a wrapper dependency is illustrated in Fig. 12. When extracting the contents of the `org.junit.jupiter:junit-jupiter` library version 5.10.3 using the Java JAR command, it is evident that the JAR file does not contain any source files; it only includes the manifest, license, and module-info files. However, the `module-info.class` file specifies the modules to be imported when this library is used as a dependency. Therefore, MAVEN will include the `junit-jupiter-api`, `junit-jupiter-params`, and `junit-jupiter-engine` dependencies when this library is defined as a dependency in a client project. To identify how many of the dependencies were used as a wrapper, we extracted all the files included under a jar file to verify if it included any source files. This analysis showed us that 16.33% of the clients had at least one direct dependency that did not contain source files. From clients that contained direct dependencies that did not contain source files, 51.89% used transitive dependency functionalities via the direct dependency.

These findings regarding transitive dependency usage in clients contribute to our understanding of why most incompatibilities during dependency updates are associated with changes in transitive dependencies.

Our results on how many transitive dependencies were related to the same group of projects as the direct dependency revealed that 35.21%, consisting of 14,680 transitive dependencies belonged to the same multi-module project a direct dependency was released under. This implies that the majority, accounting for 64.79%, which is 27,017 transitive dependencies,

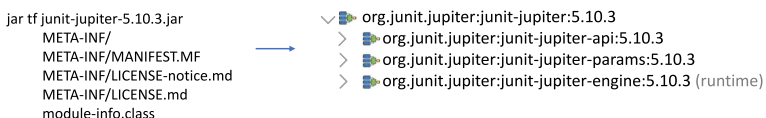


Fig. 12 `org.junit.jupiter:junit-jupiter` library version 5.10.3 which aggregates other dependencies

did not originate from a related group of projects. Transitive dependencies not related to the same group of projects could be problematic when used by client projects as they will be updated independently or migrated based on the requirements of the direct dependency declaring it.

Answering RQ5: *How often do clients directly rely on transitive dependency functionality?* The study conducted to verify the transitive dependency usage in clients showed that 61.24% of the clients had used transitive dependency functionality in their code. Notably, most of these transitive dependencies (64.79%) did not originate from the same multi-module project as the direct dependency.

3.7 RQ6: Detecting BCs Related to Transitive Dependencies

In this section, we answer **RQ6**: “Can compilation error logs alone assist in determining if the BC was related to a transitive dependency?”

3.7.1 Method

We analyzed compilation error logs to identify if they could be used to detect BC caused by a change in a transitive dependency. We used the transitive functionality calls extracted for RQ5 (methods described in Section 3.6.1) for this purpose. We created a script to divide these functionalities into package, class, method, and field and searched the logs since the fully qualified name of the class and the method or field will not appear together in the error logs. Therefore, when a BC was reported during a dependency update, we created a script to verify if any transitive dependency functionalities extracted were available in the compilation error logs. We searched if the compilation error logs included packages with the class or method names. This process is explained in Fig. 13, where the client directly invokes functions from commons-httpclient:commons-httpclient:3.1, a transitive dependency available through the direct dependency net.sourceforge.htmlunit:htmlunit:2.5. The functional call is identified using the previously described steps for detecting transitive functionality usage. This functional call is then matched with the compilation error generated when updating the net.sourceforge.htmlunit:htmlunit dependency from version 2.5 to 2.7.

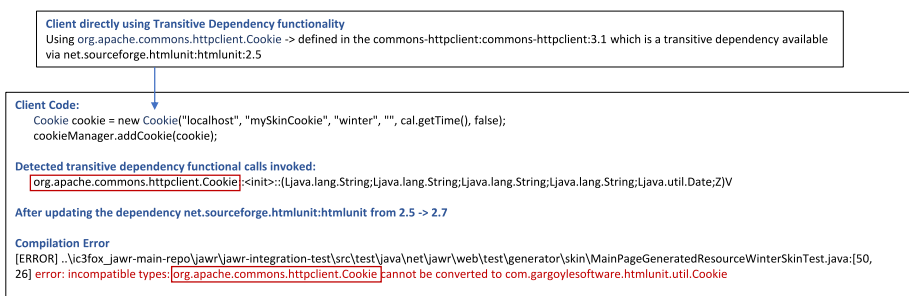


Fig. 13 Match compilation error with the transitive dependency functional call

3.7.2 Results

The automatic search through the compilation errors showed that only 3.62%, which accounted for 297 instances of the BCs, was caused due to a transitive dependency. The automated search did not report a high number as the manual analysis, which reported 20.36% of the BCs were caused by a transitive dependency change. Therefore, we randomly sampled some data from the manual analysis that reported the cause of the BC as a change in the transitive dependency but was not reported by the automatic analysis. Then, the first author manually analyzed the compilation error logs and the change in the transitive dependency to determine the reason for this low count in the automatic analysis. The highest recurring transitive change (18.57%) reported in the manual analysis was due to the migration of Java EE to Jakarta EE (Community for Java and developers 2020). This change was not recorded under our automated search as we marked all ‘javax.’ related packages as inbuilt Java packages. Therefore, these functionality calls were not reported as BCs caused due to transitive dependency changes.

Since some compilation error messages did not include the fully qualified name of the class or the method in the error log, they could not be linked with a transitive dependency functionality using the automatic analysis. These logs just reported the type of the compilation error, for example, `error: unreported exception IOException; must be caught or declared to be thrown`. Some compilation errors only included the method name and did not report the package and class to which it belonged. This finding aligns with the research conducted by Becker et al. (2019), which states that compilation error messages are not helpful in accurately reflecting the true programming change that introduced the issue. Since we only considered the compilation error logs and the functionalities of the transitive dependency for the automatic analysis, it could not accurately identify all BCs caused by transitive dependency changes. For the manual analysis, since we considered more factors mentioned in Section 3.4.1, we could identify that the cause was due to transitive dependency changes.

If the transitive dependencies introduce binary breaking changes, they may cause linkage errors at runtime, which would not be detected during compilation. Additionally, if changes in the indirect dependencies break their contract with the direct dependency, since the direct dependency is not re-compiled, this does not result in a compilation error as well.

Answering RQ6: *Can compilation error logs alone assist in determining if the BC was related to a transitive dependency?* Compilation error logs can assist in detecting BCs caused by transitive dependencies. However, we cannot solely rely on compilation error logs to determine if the BCs were caused by transitive dependencies because the compilation errors might not accurately reflect the true programming change that led to the BC.

3.8 RQ7: Discussions of Transitive Dependencies

In this section, we answer RQ7: “How do projects currently resolve BCs caused by transitive dependencies?”

3.8.1 Method

Client projects frequently rely on features provided by transitive dependencies. As such, we expect discussions around transitive dependency usage and BCs encountered due to using transitive dependencies in open-source communities. Therefore, we examined the pull requests and issues raised under popular GitHub repositories to gather insights into how transitive dependency-related BCs are handled or avoided in open-source repositories. We selected the top 100 most-starred Java repositories on GitHub as of November 2023 instead of the repositories used in the previous research questions, as they are the most popular or recognized repositories in the community and, thus, could give us more direction into using transitive dependencies.

We used the GitHub GraphQL API⁶ to extract the discussions under the pull requests and the issues in the repositories. For the discussions extracted, we first checked if they included the keywords `transitive dep`, `deep dep` and `indirect dep` in the title, description, or comments. We used ‘dep’ in the keywords as, in some discussions, dependencies were abbreviated, and using ‘dep’ would cover both dependency and dependencies. There were only 491 discussions that matched these keywords. Therefore, we randomly sampled these discussions with a confidence interval of 95% and an error rate of 5% (Wonnacott and Wonnacott 1991), which provided us a sample of 216. The first author reviewed the sampled discussions on a high level, checked for any additional keywords we could use to expand our search terms to acquire more results, and discussed them with the other authors. This process generated keywords such as `dependency hell`, `shaded`, `shading`, `shadowing`, `classpath hell`, `class path hell`, `jar hell`, `version conflicts` which we included and extracted 873 discussions matching the expanded set of keywords giving us a total of 1,364 discussions for the analysis.

First, one author went through the discussions extracted as a keyword match to determine if the discussions just mentioned the keywords during a discussion or if the discussion was actually around transitive dependencies. During this process, the discussions linked to each other were eliminated from the analysis as they discussed the same scenario with the same solution or remediation plan. Next, we followed the thematic analysis technique (Cruzes and Dyba 2011) similar to the approach followed in section 3.6.1. We labeled the data by assigning them codes based on the inferential information gathered through reading the discussions. Following a similar approach in section 3.6.1, two authors familiarized themselves with the coding by independently coding four samples at a time and then discussing the codes and repeating this process until an agreement above acceptable was reached (Zach 2021). Then, the two authors separately labeled 15% of the total discussions and calculated the inter-rater reliability. We used Cohen’s kappa to measure the inter-rater reliability, which was 0.64, indicating substantial agreement between the two authors. After that, one author labeled the rest of the issues for the qualitative analysis.

We focused on analyzing these discussions to understand how developers address BCs raised due to transitive dependencies. During the initial discussions during the coding process, we saw that some transitive dependency discussions focused on issue fixes related to BCs, and some focused on measures taken to avoid them. However, some of the discussions we analyzed only discussed BCs raised due to transitive dependencies and no direction on handling or overcoming these BCs were discussed. Additionally, some discussions were around the dependency update practices. Therefore, we excluded these types of discussions from our analysis, leaving us with 192 discussions.

⁶ <https://docs.github.com/en/graphql>

3.8.2 Results

We generated 22 codes in total for the discussions around transitive dependencies. Some codes provided direction to solve or avoid BCs raised due to transitive dependencies and provided suggestions or best practices. We excluded two codes that were specific to how Eclipse handles dependencies and source code changes related to using transitive dependency functionality, which were discussed under two separate discussions. Table 7 provides the remaining codes generated for the discussions and is further explained below.

1. Exclude Transitive Dependency The most common action suggested during these discussions was to exclude the transitive dependencies brought in by a direct dependency using the project configuration. This was also provided as a recommendation to limit the number of transitive dependencies exposed by libraries. This configuration change was mostly applied to resolve BCs raised due to transitive dependencies or as a preventative strategy. As repository

Table 7 Actions and Recommendations (codes) suggested under discussions on transitive dependencies

	Actions and Recommendations (code)	Frequency
1	Exclude Transitive Dependency	66
2	Define Transitive as a Direct Dependency	42
3	Update Direct Dependency	33
4	Change Direct Dependency Scope	10
5	Shade Classes of the Dependency	8
6	Add optional parameter true for Direct Dependency	6
7	Use functionality exposed through Transitive Dependencies	6
8	Remove Direct Dependencies	6
9	Divide functionalities into separate modules and deploy them separately	5
10	Turn additional features off by default in Libraries	4
11	Embed all required Dependencies to the deployment file	4
12	Maintaining compatibility between versions	4
13	Use a SBOM to maintain the Dependencies (both Direct and Transitive) required for the project	3
14	Define fixed version numbers for all Dependencies	3
15	Include repositories to download Transitive Dependencies under the configuration	2
16	Should not exclude all Transitive Dependencies from the configurations	2
17	Replace Direct Dependencies	1
18	Add version constraints on Transitive Dependency (Gradle)	1
19	Publish artifacts to remote repositories	1
20	Do not shade class if it is available under a Dependency	1

square/retrofit issue, 1536 suggests ‘*Actually Simple XML has transitive dependencies which are already included in Android, apparently. You need to exclude them from the converter-simplexml dependency per the “Android” section here:*’; was required to resolve a BC. This action helps to avoid having multiple versions of the same dependency in the classpath and to remove unwanted dependencies, which lead to BCs in the project. This action was illustrated under the keycloak/keycloak repository’s issue 15915 to resolve a security issue raised: ‘*I think we should be able to fix this CVE if we change the liquibase-core dependency in the parent pom.xml like that:*’

```
<dependency>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-core</artifactId>
  <version>${liquibase.version}</version>
  <exclusions>
    <exclusion>
      <groupId>org.yaml</groupId>
      <artifactId>snakeyaml</artifactId>
    </exclusion>
    <exclusion>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-text</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

However, since our analysis shows that most clients use transitive dependency functionality, excluding the transitive dependency from the configuration might not be the best action for our analyzed projects.

2. Define Transitive as a Direct Dependency The second frequent action was to define the transitive dependency as a direct dependency under the project, which will help in maintaining the correct version of the dependency required for the project similar to the comment in SeleniumHQ/selenium issue 2362 that discusses ‘... *commons-io is a transitive dependency, you shouldn’t be relying on selenium providing it for you and you should declare it as a separate library / dependency for your project.*’. Also, there were instances where this action was required as an adaptive measure when the new version of a direct dependency excluded one of its transitive dependencies, which was required by the project as mentioned in JakeWharton/butterknife repository’s pull request 562: ‘*Robolectric 3.1 removes its transitive dependency on com.google.android.support-v4:r7 so I had to include it manually.*’

3. Update Direct Dependency The third most suggested action was to update the direct dependency that brings in a transitive dependency that might cause BCs. As suggested in keycloak/keycloak repository’s issue 20319, which states ‘*Quarkus team released 3.0.3.Final a few hours ago. It is important to upgrade to fix a couple of CVEs on transitive dependencies, plus a couple of bug fixes present on this release.*’, updates were generally suggested to ensure the system avoids any security issues in the future as a preventative measure. Further updating the direct dependency could bring in the correct version of the transitive dependency that

is compatible with the project as mentioned in Repository ReactiveX/RxAndroid issue 584 *'Please consider an upgrading to the new version of RxJava which uses reactive-stream ...'*.

4. Change Direct Dependency Scope Another commonly suggested action to prevent future BCs was to change the scope of direct dependencies. By doing so, the dependency features would be restricted to specific lifecycle phases of the project. For instance, the 'test' scope restricts the dependency to test classes, and the 'provided' scope requires the dependency to be supplied by the runtime JDK or the container in which the project is deployed. A discussion that mentions this was under ReactiveX/RxJava repository's issue 154: *'Yes, its "runtime" for running the tests, but not for production usage, which is why they are marked as provided and not needed as transitive dependencies'*.

5. Shade Classes of the Dependency Shading allows you to copy and rename dependency code within your project, creating a private copy of the code. This enables direct access to those features without relying on external dependencies, thereby avoiding the addition of unnecessary transitive dependencies to the classpath. It also ensures consistent functionality throughout the project, eliminating concerns during dependency updates. This is explained in repository spring-projects/spring-boot pull request 24264, which states, *'Other projects should shade ASM themselves rather than tying themselves to Spring.'*

6. Add optional parameter true for Direct Dependency The action to set the optional parameter true when defining the dependency forces any project that requires transitive dependency functionalities to declare them as direct dependencies in the project. This would eventually be beneficial for the clients using this as a dependency as it adds fewer dependencies to the client classpath, so it will be easier to maintain the project as suggested under spring-projects/spring-framework repository's issue 8995: *'Instead of depending on commons-logging 1.1 with exclusions for every logging facility, it would be cleaner to depend on version 1.1.1, which has transitive dependencies declared as optional'*.

7. Use functionality exposed through Transitive Dependencies As an adaptive strategy, projects could use functionality exposed through transitive dependencies as it is already available in the classpath since re-declaring it will only add a new dependency version, which may lead to version conflicts. Repository iluwatar/java-design-patterns mentions this action under pull request 867: *'As i see, junit-jupiter-api is is a transitive dependency of junit-jupiter-engine. So should we remove junit-jupiter-api dependency in where we already have junit-jupiter-engine?'*.

8. Remove Direct Dependencies Removing unused direct dependencies helps maintain the project dependencies and reduces the likelihood of BCs due to transitive dependencies by reducing the number of dependencies in the classpath. An example of this action was provided as a description under eugenp/tutorials repository pull request 13568: *'Could com.baeldung.web: spring-boot-rest: 0.0.1-SNAPSHOT drop off redundant dependencies?'*

9. Divide Functionalities Into Separate Modules and Deploy them Separately This action helps both developers of the project and clients who use this project as a dependency. For developers of the project, having separate modules will allow them to declare different versions of dependencies for the different modules as required. The pull request 25 on spring-projects/spring-boot discusses this benefit *'Changes have been separated to the new projects submodule to workaround the problem of having two versions of tomcat'*

dependencies simultaneously on the classpath at compile time.' For client projects, the benefit is that when features are separated into modules, they can define the specific dependencies required in the configuration and also have the option to exclude them if necessary. This is explained under spring-projects/spring-framework issue 19081 *'I'm considering to move our Log/LogFactory variant, which is completely self-contained, to a separate spring-jcl.jar. ...'* reply *'I think spring-jcl.jar is a good idea, especially since it gives users the option to exclude it if such a need should arise.'*

10. Turn Additional Features Off by Default in Libraries When releasing new features, maintain only the core features enabled while disabling additional features in libraries that can cause BCs for clients, as suggested in issue 3392 of the alibaba/fastjson repository. *'IMO, the autoDiscover functionality should be disabled by default so that existing application logic does not break.'*

11. Embed all Required Dependencies to the Deployment File This action was suggested as a best practice to define all required dependencies in the deployment or configuration file. This ensures that all necessary dependencies are bundled with the project JAR file, benefiting clients using this project as a dependency, as explained in pull request 4302 of the Anuken/Mindustry repository. *'This task will download all dependencies, including transitive dependencies. This is very useful when packaging Mindustry for a distribution...'*

12. Maintaining Compatibility Between Versions Library developers were cautious about updating certain dependencies in the project to maintain compatibility between library versions. This was expressed under airbnb/lottie-android issues 2392 *'I can't upgrade to okio 2.0 because it would pull in Kotlin as a transitive dependency which I'd prefer not to do in case there are any apps that (for some reason) aren't using Kotlin yet.'*

13. Use a SBOM to Maintain the Dependencies(both Direct and Transitive) Required for the Project A practice mentioned for dependency management was defining a Software Bill of Materials(SBOM) file. This machine-readable inventory contains the direct and transitive dependencies in a separate POM file. This allows the project to keep track of all the required dependencies for the project and enhances its security by improving the transparency of all connected dependencies (Xia et al. 2023). Their research indicates that while SBOMs bring transparency to software projects by enabling accountability and security, they are not yet widely adopted in OSS, and there is a lack of maturity in SBOM tooling. However, we anticipate that the use of SBOMs will increase due to software compliance mandates by the US government and the EU, with other governments expected to follow (Biden 2021; European Commission 2024). The benefit of using SBOMs for dependency management is explained under square/retrofit issues 3231 *'Notice how you just leave the versions off when you use a BOM. This can help reduce the number of dependency compatibility surprises one can encounter, especially if a transitive dependency brings in a newer version of one of the components (it'll be reduced to the BOM's version).'*

14. Define fixed version number for all Dependencies It is advisable to specify a fixed version number for all project dependencies. If a version is not defined, the dependency resolution mechanism will default to the latest available version. However, the latest version available at the time of development may not remain compatible as the library evolves. Also Discussed under OpenAPITools/openapi-generator repository's pull request 13593 *'I believe, the problem is with the pom template(s) missing a version field for the spring-boot-maven-*

plugin, which causes maven to pick 3.0.0, which again has transitive dependencies built for java 17.' Further, defining a fixed version for a dependency, rather than using a version range, is preferable when aiming to avoid conflicting versions within that range. By specifying a particular version, you ensure the project uses the one that is compatible.

15. Include Repositories to Download Transitive Dependencies Under the Configuration

To avoid downloading transitive dependencies from different remote repositories and to prevent dependency resolution problems, it is advisable to configure a specific repository for downloading these dependencies. This issue was discussed under apache/skywalking issues 6867 '*... we still have some (transitive) dependencies that downloaded from jcenter, we should find them out and point the download address to maven central.*'

16. Should not Exclude All Transitive Dependencies from the Configurations When managing dependencies, it is not advisable to exclude all transitive dependencies from the configurations, as some are essential for the linkage and runtime of the projects. This benefit of having transitive dependencies was discussed in airbnb/lottie-android issue 1538: '*I think somehow in your build, you are excluding the okio transitive dependency. Make sure you are not excluding okio and this problem should go away.*'

17. Replace Direct Dependencies Replacing a direct dependency with another was recommended when an alternative library offering the same functionality was available and provided greater benefits to the project than the previously used dependency. Discussed under spring-projects/spring-framework repository issue number 8333. '*The problem can be resolved in one of three ways: ... (Better) Replace commons-logging.jar with jcl104-over-slf4j.jar (See <http://www.slf4j.org/manual.html#gradual>)*'

18. Add Version Constraints on Transitive Dependency (Gradle) Including dependency constraints allows us to define a version or version range for both direct and indirect dependencies. This method is preferred for applying constraints to all dependencies within a configuration to avoid conflicting versions or versions with vulnerabilities or bugs. The suggestion was discussed under OpenAPITools/openapi-generator issue number 14901 '*You can constrain the version of transitive dependencies in gradle.*'

19. Publish Artifacts to Remote Repositories In general, all libraries should publish their artifacts to a remote repository for client use. This request was specifically made to avoid dependency resolution problems in the future, as the library in question was going to be removed. Discussed under elastic/elasticsearch repository's pull request 68926 '*Eventually this library is going to go away entirely and it's actually not really required at runtime by any of these transitive dependencies, but to ensure we don't incidentally introduce any breaking dependencies in the meantime the safest thing is just to publish the artifact to address the dependency resolution errors.*'

20. Use Functions from Jar Dependency Instead of Shading This action was recommended by clients using a library, requesting the library to use a direct dependency API functional call instead of shading it. This is because, when a class is already available on the class path and another shaded version exists, it can cause issues during runtime resolution. Therefore, a client of OpenAPITools/openapi-generator raised this as a concern under issue 3034. '*We would like an unshaded version of the jar to be available so we can depend on it without including duplicate dependencies in our class path.*'

Overall, some of these actions, such as defining a transitive dependency as a direct dependency under the project or encouraging the developers to do so, might resolve issues caused by BCs for that scenario but might introduce issues such as dependency hell if too many dependencies are defined under the project.

Answering RQ7: *How do projects currently resolve BCs caused by transitive dependencies?* According to our analysis of open-source discussions on transitive dependencies, the most common actions to resolve issues due to transitive dependencies are excluding transitive dependencies, defining the transitive dependency as a direct dependency, and updating the direct dependency that brings in the transitive dependency.

4 Discussion

In this section, we present the implications of the results based on our findings and threats to validity.

4.1 Implications of the Study

We can derive several practical implications from our study from the perspective of library developers, client developers, and researchers.

4.1.1 Implications for Library Developers

Library developers should be vigilant when applying changes during non-MaJor updates, as these versions should not introduce BCs. However, our study reports under RQ4 that almost half of the BCs were introduced in non-MaJor updates.

Our analysis on RQ5 showed that some libraries are made available as a package aggregation artifact that bundles all compatible components required for the functionality. Thus, this could be a potential pattern for libraries part of a multi-module project, allowing them to be released as one library that client projects will use. This will prevent clients from facing incompatibilities during dependency updates, as they will only be required to update one library version.

4.1.2 Implications for Client Developers

Client developers should think twice before using the functionality of transitive dependencies. As our results indicate, under RQ5, more than half of the clients directly use transitive dependency functionality, and most of these used transitive dependencies did not belong to the same multi-module project as the direct dependency. These transitive dependencies will be updated independently under the direct dependency and can also be removed in future releases. Thus, the use of transitive dependencies often leads to many BCs during dependency updates, and these BCs were most common during non-MaJor dependency updates.

Moreover, our findings indicate that the impact of a BC on clients varies based on how the breaking functionality is used in their code. Therefore, when addressing incompatibilities related to BCs, client developers must consider the context in which the functionality is used. When applying dependency updates in isolation, careful consideration is needed to determine whether the updated dependency relies on another dependency or vice versa. This is crucial because updating a dependency in isolation may lead to incompatibilities with other dependencies in the artifact. Further clients should be cautious when using transitive dependencies which are not related to the same multiple module project as their changes are released separately and the direct dependency can decide to migrate from that transitive library to another library. These changes could lead to BCs in the client projects.

Clients should focus on updating dependencies regularly as our results show that most dependencies in open source projects are not kept up-to-date. If the dependencies are not updated regularly the clients might not fully utilize the libraries' features, and it could include potential vulnerable code, which might get resolved in later versions.

4.1.3 Implications for Researchers

Investigating techniques to capture clients' use of transitive dependency functionalities and their contributions to BCs during dependency updates is a much-needed research area. Tools should be developed to detect the use of transitive dependencies so that clients are aware that they are using functionalities of dependencies that are not defined directly under their project configurations. This would help reduce the risk of encountering BCs due to transitive dependencies during updates. Our results indicate that changes in transitive dependencies are a significant factor in introducing BCs during dependency updates; hence, analyzing all dependencies connected to a client project when updating a dependency will provide a comprehensive understanding of all potential BCs that impact a client project.

Current static analysis tools offer a lower granularity level in reporting library changes, such as whether a field, method, or class was removed. However, our findings on RQ3 reported that removing an entire API package was the second most common BC between library versions, and this was not clearly indicated by the tool since it focused on lower-level granularity. Therefore, these tools can be improved to report BCs at a more appropriate level of granularity, which will help project maintainers better understand and address BCs.

4.2 Catalog of Changes to Remediate and Avoid BCs Due to Transitive Dependencies

We built a catalog of changes based on the actions suggested in handling and preventing the BCs raised by transitive dependency changes. The *Catalogue of Refactoring* site (Fowler *n.d.*) proposed by Fowler and Beck (1997) using the pattern language structure inspired us to develop this structure. The catalog of changes we propose will assist developers in avoiding BCs and will remediate current BCs that the developers experience due to transitive dependencies. The change catalog provides actions recommended for client and library projects separately.

In the diagrams, 'C' represents a client. 'D' within in a continuous circle denotes a direct dependency, while 'D' within a dotted circle denotes a transitive dependency. Arrows indicate connections to a project or suggest inclusion or usage. Text, arrows, or lines in green indicate

that the elements should be included, whereas those in red indicate that the elements should be removed or avoided.

4.2.1 Changes proposed for Client Projects

Exclude Transitive Dependency Figure 14a represents the change of excluding the transitive dependency. This change can be applied by changing the configuration file, which includes the dependency information. For a MAVEN project, this change will be included in the POM.xml file. Listing 2 illustrates how a transitive dependency can be excluded from a MAVEN project, which would prevent it from being included in the project classpath.

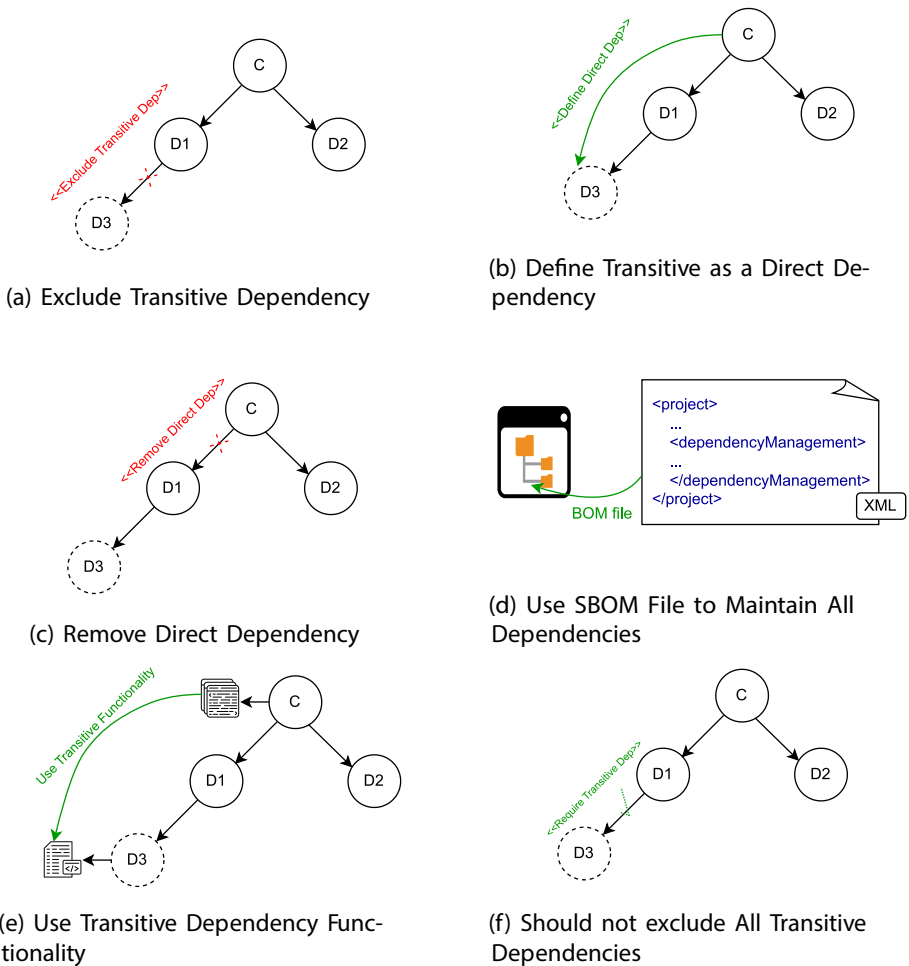


Fig. 14 Catalog of Changes For Client Projects


```
<dependency>
  <groupId>org.yaml</groupId>
  <artifactId>snakeyaml</artifactId>
  <scope>compile</scope>
  <version>1.30</version>
  <exclusions>
    <exclusion>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-text</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Listing 2 A Dependency Block after excluding a Transitive Dependency for the client projects from the pom.xml file

Define Transitive as a Direct Dependency Figure 14b illustrates defining a transitive dependency as a direct dependency for the project. The dependencies should be included in the project configuration file. This will allow the project to define the correct dependency version compatible with the project.

Remove Direct Dependency Figure 14c shows removing a direct dependency declared under the project configuration. The direct dependencies recommended for removal are typically either unused by the project or may already be available in the required version as a transitive dependency. Removing unnecessary dependencies will reduce the total number of dependencies included in the project classpath.

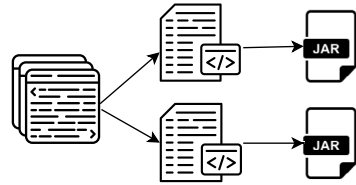
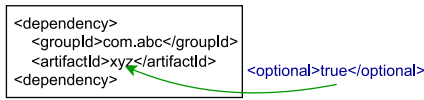
Use SBOM File to Maintain All Dependencies Figure 14d shows that all required dependencies, including direct and transitive dependencies, can be maintained using a Software Bill Of Materials (SBOM) file. Using a single location to manage all dependencies improves the project's maintainability.

Use Transitive Dependency Functionality Figure 14e illustrates that client projects should use transitive dependencies when appropriate, without defining them as direct dependencies. This approach is highly recommended for dependencies within the same multi-module project or for dependencies that fall under a wrapper dependency.

Should not exclude All Transitive Dependencies Figure 14f shows that certain transitive dependencies are necessary during linkage or runtime for the direct dependency. Therefore, excluding all transitive dependencies from a project is not advisable.

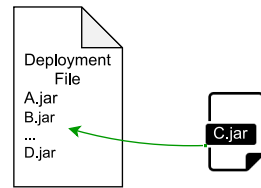
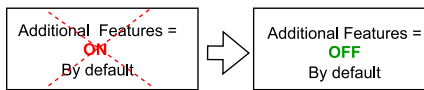
4.2.2 Changes proposed for Library Projects

Include Optional Parameter for Direct Dependency Figure 15a displays that the optional parameter can be included in the direct dependency configuration. Adding the optional parameter would prevent the dependency from being included in the classpath of the clients using it.



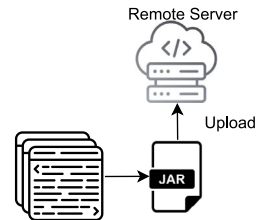
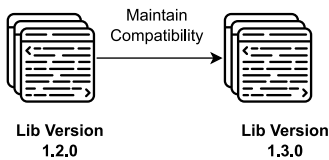
(a) Include Optional Parameter for Direct Dependency

(b) Divide Functionality into Separate Modules



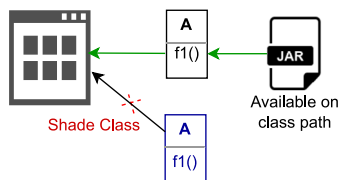
(c) Turn Additional Features Off by Default

(d) Embed All Required Dependencies to the Jar File



(e) Maintaining compatibility between two library versions

(f) Publish artifacts to remote repositories



(g) Do not shade class if it is available under a dependency

Fig. 15 Catalog of Changes For Library Projects

Divide Functionalities to Separate Modules and Deploy them Separately Figure 15b illustrates that functionalities that can be isolated should be extracted as separate modules, adhering to the principle of separation of concerns. This enhances project maintainability. These modules can then be deployed as independent JAR files, allowing clients to rely on only the necessary JARs.

Turn Additional Features Off by Default Figure 15c illustrates that the features additional to the core functionality of the libraries must be turned off by default. Then, the clients who need those additional features can turn them on as required for their projects, and this would not cause BCs for clients not needing the functionality.

Embed All Required Dependencies to the Jar File Figure 15d displays that the jar file should include all dependencies required for the functionality of the project. This ensures that the client projects using these libraries would not encounter BCs related to missing dependencies necessary for using the library.

Maintaining Compatibility Between Two Library Versions Figure 15e highlights the importance of library developers maintaining compatibility between versions to avoid introducing BCs.

Publish Artifacts to Remote Repositories Figure 15f shows that libraries should upload their artifacts to remote repositories for client projects to access.

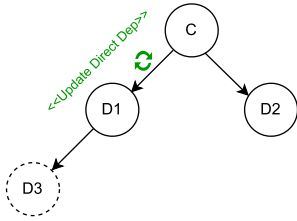
Do not Shade Class if it is Available Under a Dependency Figure 15g displays that if the dependency is already available under the class path use the dependency functionality without shading it.

4.2.3 Changes Proposed for Both Client and Library Projects

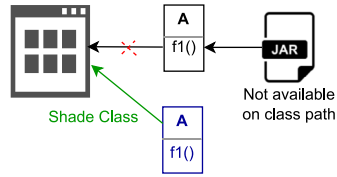
Update Direct Dependency Figure 16a demonstrates the update of the direct dependency of the project. The primary motivation behind this action was to update direct dependencies containing vulnerabilities for security purposes. This would allow the transitive dependency to be updated indirectly as well.

Shade Classes of the Dependency Figure 16b illustrates that the dependency classes required for the project will be shaded (copied) under the project itself. This will allow the project to access the shaded class functionality directly and not depend on the dependency class functionality. For client projects, this would be helpful to avoid compatibility issues with defining new dependencies, and for libraries, it will reduce the number of dependencies introduced to the client's classpath.

Change Direct Dependency Scope Figure 16c shows that the scope of the direct dependency should be changed. This was suggested mainly for client projects to keep the dependencies required for testing purposes separate from the compile and runtime dependencies. For library projects, it was advised to use the 'provided' scope for the dependencies to prevent it being included in the client's classpath.



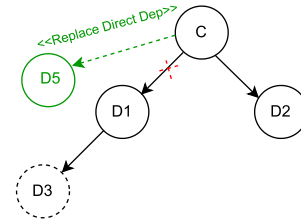
(a) Update Direct Dependency



(b) Shade Classes of the Dependency

```
<dependency>
<groupId>com.abc</groupId>
<artifactId>xyz</artifactId>
<scope>test</scope>
</dependency>
```

(c) Change Direct Dependency Scope



(d) Replace Direct Dependencies

```
<project>
...
</project>
```

```
<repositories>
<repository>
<id>XX</id>
<url>http://maven/abc</url>
</repository>
</repositories>
```

(e) Include Repositories to Download Transitive Dependency

```
<dependency>
<groupId>com.abc</groupId>
<artifactId>xyz</artifactId>
<version>[1.0,2.0]</version>
</dependency>
```

```
<version>1.6</version>
```

(f) Define fixed version numbers for all Dependencies

```
dependencies {
  implementation 'org.abc:xyz'
}
```

```
constraints {
  implementation('org.abc:xyz:4.5.3') {
    because 'previous versions have a bug impacting this application'
  }
}
```

(g) Add version constraints on Transitive Dependency

Fig. 16 Catalog of Changes For Both Clients and Library Projects

Replace Direct Dependencies Figure 16d shows that, depending on the requirements, it may be more beneficial to replace the current direct dependency with an alternative dependency.

Include Repositories to Download Transitive Dependencies Under the Configuration Figure 16e illustrates the code block that should be included in the project configuration file if a direct or transitive dependency needs to be downloaded from a specific repository

rather than Maven Central. Alternatively, the repository can be also defined in the Maven 'settings.xml' file.

Define Fixed Version Numbers for all Dependencies Figure 16f demonstrates that projects should specify a version for all defined dependencies. Additionally, it is recommended to use a fixed version rather than a range of versions.

Add Version Constraints on Transitive Dependency Figure 16g displays that constraints on both direct and transitive dependencies can be specified in the configuration file based on the project requirements.

All actions, except for "Change Direct Dependency Scope", "Add optional parameter true to direct dependency," and "Add constraints on transitive dependency versions" are generic and applicable to other programming languages. Dependency scopes in Java Maven and Gradle differ from those in other languages, and the optional parameter may not be available in other languages. Therefore, these three actions are more specific to Maven-built or Gradle-built Java projects. The configuration changes proposed to resolve transitive dependency issues could introduce new issues if required dependencies are excluded from the classpath or if introducing new dependencies results in version conflicts. Thus, developers should be cautious when implementing these changes.

4.3 Threats to Validity

A threat to construct validity lies in the approach used to select successfully compiled clients for the study. Since some repositories contained multiple modules which were considered as clients, and these clients relied on the binaries of the other clients under the same repository we could not use the `Maven compile` command on individual clients. Instead, we had to use the `Maven install` command on the clients to generate the required binaries. Therefore, clients that compiled successfully but failed to be packaged into executables were excluded. Hence, we could have disregarded some BCs existing in these clients. Adding to this construct validity, we selected the repositories for analysis using the 2020 Libraries.io dataset and began the experiments in 2021. Consequently, most projects in our dataset have been active in recent years. There are some projects with significant periods of inactivity as well as projects with a small number of stars. Future work can validate our findings using a more recent dataset.

Another threat to construct validity is not extracting some transitive dependency calls through our analysis. For instance, when a client class (A) inherits a superclass (B) from its direct dependency, and this superclass (B) inherits a superclass (C) from its direct dependency, which is transitive to the client. Our analysis did not cover that granularity of transitive dependency functionality extraction. Therefore, if the class C from the transitive dependency was removed or changed in the above instance, this could introduce BCs on the client end. Consequently, we might have underestimated the impact of transitive dependency changes.

Another construct validity is that when extracting the transitive dependency functional calls, we excluded the functional calls belonging to the 'javax' package as we considered those functionalities linked to the inbuilt Java functionality. However, this exclusion prevented us from automatically capturing the migration of Java EE to Jakarta EE, when using the compilation error logs to identify BCs introduced due to transitive dependencies.

For the analysis of detecting BCs, we included both the library internal APIs used by clients and the library APIs that were deprecated in previous releases and then removed, leading to BCs. This is another construction validity. We considered them as part of the analysis because, during dependency updates, client developers would encounter the same BC impact regardless of whether the APIs are internal or deprecated. However, it would be interesting to analyze them separately.

A threat to the conclusion validity of our research could have occurred when labeling the cause of the BCs and the discussions around transitive dependencies. However, we tried to mitigate this threat by involving two coders and calculating the inter-rater reliability of the labeled data.

External validity to the study is in generalizing the results of this study to other programming languages. We focused on more than 18,000 Java projects built based on the MAVEN built tool that altogether used more than 7,000 unique Java libraries. As with any other empirical research in this area (Ochoa et al. 2022b; Raemaekers et al. 2017; Harrand et al. 2022), our findings cannot be generalized across other programming languages. Additionally, we compiled these projects using Java 8 and Java 11 and did not consider other Java versions or Java itself as a dependency. Our results on BCs reported that only 3.45% were due to a Java version incompatibility; therefore, including additional Java versions would have kept our findings relatively the same.

5 Related Work

We reviewed the most closely related work on library evolution, the BCs introduced, and its impact on clients. According to our knowledge we are the first to conduct a large-scale empirical study covering the impact of both source and binary BCs on clients.

Library Evolution and Breaking Changes How libraries evolve, their practices, and stability is a much researched area (Dig and Johnson 2006; Koçi et al. 2019; Møller and Torp 2019; Xavier et al. 2017). Dig and Johnson (2006) analyzed the API changes and their compatibility to introduce that API changes comprise breaking and non-breaking changes. They introduced a catalogue of BCs and showed that refactoring contributes to 80% of the breaking API changes. In another study conducted on Java APIs, Dietrich et al. (2014) report that 75% of the adjacent library versions are incompatible. Jezek et al. (2015) continued this study and reported that 80% of the adjacent library versions break compatibility. Further adding to these findings, Xavier et al. (2017) reported that more popular and active libraries tend to contain more BCs in their releases. These studies demonstrate that BCs are common under library releases, which leads us to study syntactic BCs in this research.

Given that BCs can lead to failures in client projects, it is crucial to inform clients when such changes are introduced. The recommended practice is to document BCs in changelogs and release notes (Brito et al. 2018b, 2020). However, existing research indicates that documentation for less than 50% of BCs is available (Koçi et al. 2019). Considering this, clients face uncertainty regarding when it is safe to update their dependencies. The semantic versioning scheme allows library developers to signal when BCs are introduced releases. Yet, studies show that libraries often violate the semantic versioning principles and introduce BCs even in non-Major releases (Raemaekers et al. 2014, 2017; Jezek et al. 2015; Dietrich et al. 2019). Our research aligns with these findings, presenting evidence that BCs are indeed introduced in `Minor` and `Patch` releases. It is noteworthy that while previous

studies focused on BCs between library versions, our approach uniquely assesses the impact of these BCs on client projects.

Breaking Change Impact on Clients Recent research has analyzed the impact of syntactic BCs have on clients when updating their dependencies (Bavota et al. 2013; Jezek et al. 2015; Xavier et al. 2017; Raemaekers et al. 2017; Ochoa et al. 2022b). Xavier et al. (2017) reported 2.54% of total clients are impacted by BCs, while (Bavota et al. 2013) concluded that around 5% of the total source code in client projects are impacted by BCs. However, these studies determined if the client uses the BC functionality by analyzing the import statements of a class. This could both overestimate the impact of certain BCs as well as underestimate the impact as not all BC functionality is available as direct imports in classes.

Raemaekers et al. (2017) reported that binary BCs significantly impact clients by injecting each binary BC one at a time. However, applying BCs in isolation will overestimate the impact as BCs could rely on other changes to the code and might not cause BCs if updated together with all changes. Ochoa et al. (2022b) replicated the prior study and reported that 7.9% of the clients are impacted due to binary BCs. Both Raemaekers et al. (2017) and Ochoa et al. (2022b) focused only on binary BCs, and additionally, our research analyzed the impact of both source and binary BCs on clients. Furthermore, all of the previous researchers conducted their research by analyzing the impact of the adjacent version available for the library, which does not create a realistic scenario when updating dependencies. Thus, we updated the libraries to the latest stable version when conducting our research.

BC Detection Tools BC detection tools use static analysis techniques to detect BCs between library versions. Some of these tools include CLIRR⁷, JAPICMP⁸, JAPICHECKER, JAPITOOL⁹, SIGTEST¹⁰, APIDIFF (Brito et al. 2018a) and REVAPI¹¹ which consider both source and binary code analysis to detect BCs introduced between two library versions. Notably, all these tools report the potential BCs introduced from the library perspective, which can contain many false positives for clients utilizing these libraries as they only use a much smaller subset of these reported BCs. Ochoa et al. (2022b) designed Maracas, a tool to automatically detect breaking declarations in a new library release used in client code through static analysis. However, Maracas has limitations in detecting breaking declarations implemented through inheritance hierarchies, overridden methods, and exception handling that affect its ability to identify BCs comprehensively. In contrast, our approach reports BCs without relying on the outcomes of static analysis tools, thereby overcoming the limitations associated with existing tools and distinguishing our work from the previous study (Ochoa et al. 2022b). We discovered that changes in transitive dependencies contributed significantly to introducing BCs to clients. However, these static analysis tools used to detect BCs will not detect the BCs introduced by transitive dependencies as they apply their analysis on multiple versions of the same library and will not consider the libraries they depend on.

Transitive Dependencies Exploring transitive dependency usage and its implications in introducing BCs has received limited attention from researchers. Some studies report transitive dependencies can introduce vulnerabilities in clients (Pashchenko et al. 2018; Prana

⁷ <https://clirr.sourceforge.net/>

⁸ <https://github.com/siom79/japicmp>

⁹ <https://packages.debian.org/stretch/devel/japitools>

¹⁰ <http://wiki.apidesign.org/wiki/SigTest>

¹¹ <https://revapi.org/revapi-site/main/index.html>

et al. 2021; Düsing and Hermann 2022; Pashchenko et al. 2022), and researchers (Kula et al. 2014; Han et al. 2020) have highlighted that it is an essential future work to understand how transitive dependencies contribute to the evolution of libraries. Yet, there is a notable absence of research on how transitive dependency changes contribute to syntactic BCs in clients during dependency updates, and since our findings indicated that changes in transitive dependencies significantly impact clients, we were inspired to investigate further about transitive dependencies.

6 Conclusion

This paper presents an empirical analysis of the impact of BCs on client projects while updating their dependencies, using 142,355 direct dependencies declared in 18,415 clients.

The study revealed that a significant portion of the dependencies in clients, precisely 71.60%, were not up-to-date with the latest available versions for the libraries, and these outdated dependencies were distributed across 43.79% of the clients analyzed. When updating these outdated dependencies to their latest versions, 11.58% encountered failures due to BCs impacting the artifacts, and almost half of these BCs were introduced during a non-Major update, violating the semantic versioning principle. The most common change within the library code that led towards a BC was changing the result type of a method. The most common change in the library that led to client incompatibilities is changes in transitive dependencies, which was supported by the results that 61.24% of the clients use transitive dependency functionalities. Most compilation errors caused by a BC will not assist in determining if a transitive dependency change caused it. Even though most projects use transitive dependency functionality, the software engineering community's recommended practice to avoid BCs caused by transitive dependencies is to exclude them from the project configuration.

For future work, we aim to broaden our research by including Java projects using Gradle as their build tool. Additionally, we intend to apply similar methodologies to other statically typed programming languages to detect the impact of BCs introduced by their respective libraries. Further, we will analyze the impact of behavioral BCs on client projects, as they have received limited attention in current research, and will help provide the overall impact of BCs on clients during dependency updates.

Acknowledgements This work was supported by the Marsden Fund Council from Government funding, administered by the Royal Society Te Apārangi, New Zealand. The work of the fifth author was supported by a gift by Oracle Labs Australia. In addition, the authors wish to acknowledge the Centre for eResearch at the University of Auckland for their help in facilitating this research (<http://www.eresearch.auckland.ac.nz>).

Data Availability Statement All study results, data, and scripts are available at Jayasuriya et al. (2024b).

Declarations

Conflict of interest The authors declare that one of the authors is an editor at the journal and we have no other conflicts of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is

not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.


References

- Alfadel M, Costa DE, Shihab E (2021) Empirical analysis of security vulnerabilities in python packages. In: International Conference on Software Analysis, Evolution and Reengineering (SANER '21). IEEE, pp 446–457. <https://doi.org/10.1109/SANER50967.2021.00048>
- Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2013) The evolution of project inter-dependencies in a software ecosystem: the case of apache (ICSM '13). IEEE, pp 280–289. 9780769549811. <https://doi.org/10.1109/ICSM.2013.39>
- Becker BA, Denny P, Pettit R, Bouchard D, Bouvier DJ, Harrington B, Kamil A, Karkare A, McDonald C, Osera P-M, Pearce JL, Prather J (2019) compiler error messages considered unhelpful: the landscape of text-based programming error message research. In: Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (Aberdeen, Scotland UK) (ITiCSE-WGR '19). Association for Computing Machinery, New York, NY, USA, pp 177–210. 9781450375672. <https://doi.org/10.1145/3344429.3372508>
- Biden JR Jr (2021) Executive Order on Improving the Nation's Cybersecurity. <https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/>
- Brito A, Valente MT, Xavier L, Hora A (2020) You broke my code: understanding the motivations for breaking changes in APIs. *Emp Softw Eng* 25(2):1458–1492. <https://doi.org/10.1007/s10664-019-09756-z>
- Brito A, Xavier L, Hora A, Valente MT (2018a) APIDiff: detecting API breaking changes. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18). IEEE, pp 507–511. <https://doi.org/10.1109/SANER.2018.8330249>
- Brito A, Xavier L, Hora A, Valente MT (2018b) Why and how Java developers break APIs. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18). IEEE, pp 255–265. <https://doi.org/10.1109/SANER.2018.8330214>
- Bruneton E, Kuleshov E, Loskutov A, Forax R (2022) ASM. <https://asm.ow2.io/>
- Cox J, Bouwers E, van Eekelen M, Visser J (2015) Measuring dependency freshness in software systems. In: International Conference on Mobile Software Engineering and Systems (MOBILESoft '15). IEEE, pp 109–118. <https://doi.org/10.1109/ICSE.2015.140>
- Cruzes DS, Dyba T (2011) Recommended steps for thematic synthesis in software engineering. In: 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp 275–284. <https://doi.org/10.1109/ESEM.2011.36>
- Dann A, Hermann B, Bodden E (2023) UPCY: safely updating outdated dependencies. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp 233–244. <https://doi.org/10.1109/ICSE48619.2023.00031>
- Darcy J (2021) Kinds of compatibility. <https://wiki.openjdk.org/display/csr/Kinds+of+Compatibility>
- Decan A, Mens T (2021) What do package dependencies tell us about semantic versioning? *IEEE Trans Softw Eng* 47, 6 (6 2021), 1226–1240. 19393520. <https://doi.org/10.1109/TSE.2019.2918315>
- des Rivières J (2017) Evolving Java-based APIs 2. https://wiki.eclipse.org/Evolving_Java-based_APIs_2
- Dietrich J, Jezek K, Brada P (2014) Broken promises: an empirical study into evolution problems in Java programs caused by library upgrades. In: Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE 14). pp 64–73. <https://doi.org/10.1109/CSMR-WCRE.2014.6747226>
- Dietrich J, Jezek K, Brada P (2016) What java developers know about compatibility, and why this matters. *Empirical Softw Engg* 21, 3, 1371–1396. 1382-3256. <https://doi.org/10.1007/s10664-015-9389-1>
- Dietrich J, Pearce D, Stringer J, Tahir A, Blincoe K (2019) Dependency versioning in the wild. In: 16th International Conference on Mining Software Repositories (MSR '19). pp 349–359. <https://doi.org/10.1109/MSR.2019.00061>
- Dig D, Johnson R (2006) How Do APIs Evolve? A Story of Refactoring: Research Articles. *J Softw Maintenance and Evol: Res Pract* 18, 2, 83–107. 1532-060X <https://doi.org/10.1002/smr.328>
- Düsing J, Hermann B (2022) Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. *Digital Threats* 3, 4, Article 38, 25 pages. <https://doi.org/10.1145/3472811>
- El Emam K (1999) Benchmarking Kappa: Interrater agreement in software process assessments. *Empir Softw Eng* 4(1999):113–133

- European Commission (2024) EU Cyber Resilience Act. <https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act>
- Foo D, Chua H, Yeo J, Ang MY, Sharma A (2018) Efficient static checking of library updates. In: 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). ACM, pp 791–796. 9781450355735. <https://doi.org/10.1145/3236024.3275535>
- Fowler M (n.d.) Catalog of Refactorings. <https://refactoring.com/catalog/>
- Fowler M, Beck K (1997) Refactoring: Improving the design of existing code. In: 11th European Conference. Jyväskylä, Finland
- Gosling J, Joy B, Steele G, Bracha G, Buckley A, Smith D, Bierman G (2021) The Java language specification. Oracle America, Inc
- Han J, Deng S, Lo D, Zhi C, Yin J, Xia X (2020) An empirical study of the dependency networks of deep learning libraries. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp 868–878. <https://doi.org/10.1109/ICSME46990.2020.00116>
- Harrand N, Benelallam A, Soto-Valero C, Bettega F, Barais O, Baudry B (2022) API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages. *J Syst Softw* 184(2022), 111134. 0164-1212. <https://doi.org/10.1016/j.jss.2021.111134>
- He H, He R, Gu H, Zhou M (2021) A Large-Scale Empirical Study on Java Library Migrations: Prevalence, Trends, and Rationales (ESEC/FSE '21). ACM, pp 478–490. 9781450385626. <https://doi.org/10.1145/3468264.3468571>
- Inc Tidelift (2022) Libraries.io - The Open Source Discovery Service. <https://libraries.io/data>
- Java Community for Java and JVM developers (2020) Transition from Java EE to Jakarta EE. <https://blogs.oracle.com/javamagazine/post/transition-from-java-ee-to-jakarta-ee>
- Jayasuriya D, Terragni V, Dietrich J, Ou S, Blincoe K (2023) Understanding Breaking Changes in the Wild. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (, Seattle, WA, USA.) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, pp 1433–1444. 9798400702211. <https://doi.org/10.1145/3597926.3598147>
- Jayasuriya D, Terragni V, Dietrich J, Blincoe K (2024a) Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications. *Proc. ACM Softw. Eng.* 1, FSE, Article 56, 24 pages. <https://doi.org/10.1145/3643782>
- Jayasuriya D, Terragni V, Dietrich J, Ou S, Blincoe K (2024b). Replication Package For An Extended Study of Syntactic Breaking Changes in the Wild. <https://doi.org/10.5281/zenodo.7978506>
- Jezek K, Dietrich J, Brada P (2015) How Java APIs Break - An Empirical Study. 65, C, pp 129–146. 0950-5849 <https://doi.org/10.1016/j.infsof.2015.02.014>
- Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The Promises and Perils of Mining GitHub. In: 11th Working Conference on Mining Software Repositories (Hyderabad, India) (MSR 2014). Association for Computing Machinery, New York, NY, USA, pp 92–101. 9781450328630. <https://doi.org/10.1145/2597073.2597074>
- Kikas R, Gousios G, Dumas M, Pfahl D (2017) Structure and Evolution of Package Dependency Networks. In: 14th International Conference on Mining Software Repositories (MSR '17). IEEE, pp 102–112. 9781538615447. <https://doi.org/10.1109/MSR.2017.55>
- Koçi R, Franch X, Jovanovic P, Abelló A (2019) Classification of Changes in API Evolution. In: 23rd International Enterprise Distributed Object Computing Conference (EDOC '19). IEEE, pp 243–249. <https://doi.org/10.1109/EDOC.2019.00037>
- Kula RG, De Roover C, German D, Ishio T, Inoue K (2014) Visualizing the evolution of systems and their library dependencies. In: 2014 Second IEEE Working Conference on Software Visualization. pp 127–136. <https://doi.org/10.1109/VISSOFT.2014.29>
- Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? *Emp Softw Eng* 23, 1, 384–417. 1382-3256. <https://doi.org/10.1007/s10664-017-9521-5>
- Mohagheghi P, Conradi R, Killi OM, Schwarz H (2004) An empirical study of software reuse vs. defect-density and stability. In: Proceedings of the 26th International Conference on Software Engineering (ICSE '04). IEEE Computer Society, USA, pp 282–292. 0769521630
- Møller A, Nielsen BB, Torp MT (2020) Detecting Locations in JavaScript Programs Affected by Breaking Library Changes. *Proc. ACM Program. Lang.* 4:1–25. <https://doi.org/10.1145/3428255>
- Møller A, Torp MT (2019) Model-based testing of breaking changes in node.js libraries. In: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019). ACM, pp 409–419. 9781450355728. <https://doi.org/10.1145/3338906.3338940>

- Mujahid S, Abdalkareem R, Shihab E, McIntosh S (2020) Using Others' Tests to Identify Breaking Updates. In: 17th International Conference on Mining Software Repositories (MSR '20). ACM, pp 466–476. 9781450375177. <https://doi.org/10.1145/3379597.3387476>
- Ochoa L, Degueule T, Falleri J-R (2022a) BreakBot. In ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '22). ACM. <https://doi.org/10.1145/3510455.3512783>
- Ochoa L, Degueule T, Falleri J-R, Vinju J (2022b) Breaking bad? Semantic versioning and impact of breaking changes in maven central: an external and differentiated replication study. *Empirical Softw. Engg.* 27, 3, 42 pages. 1382-3256. <https://doi.org/10.1007/s10664-021-10052-y>
- O'Connor C, Joffe H (2020) Intercoder Reliability in Qualitative Research: Debates and Practical Guidelines. *International Journal of Qualitative Methods* 2020:160906919899220. <https://doi.org/10.1177/1609406919899220>
- Olivera FR (2022) MVN Repository: repository stats. <https://mvnrepository.com/repos>
- Oracle (n.d.) Java Virtual Machine Specification: Chapter 5. Loading, Linking, and Initializing. <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html>
- Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2018) Vulnerable Open Source Dependencies: Counting Those That Matter. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18). ACM. 9781450358231. <https://doi.org/10.1145/3239235.3268920>
- Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2022) Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Trans Softw Eng* 48(5):1592–1609. <https://doi.org/10.1109/TSE.2020.3025443>
- Prana GAA, Sharma A, Shar LK, Foo D, Santosa AE, Sharma A, Lo D (2021) Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empir Softw Eng* 26(2021):1–34
- Preston-Werner T (n.d.) Semantic Versioning 2.0.0. <https://semver.org/>
- Raemaekers S, van Deursen A, Isser J, (2017) Semantic versioning and impact of breaking changes in the Maven repository. *J Syst Softw* 129(140–158):0164–1212. <https://doi.org/10.1016/j.jss.2016.04.008>
- Raemaekers S, van Deursen A, Visser J (2014) Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In: 14th International Working Conference on Source Code Analysis and Manipulation (SCAM '14). IEEE, pp 215–224. <https://doi.org/10.1109/SCAM.2014.30>
- Salza P, Palomba F, Di Nucci D, D'Uva C, De Lucia A, Ferrucci F (2018) Do Developers Update Third-Party Libraries in Mobile Apps? (ICPC '18). Association for Computing Machinery, New York, NY, USA, pp 255–265. 9781450357142 <https://doi.org/10.1145/3196321.3196341>
- Stringer J, Tahir A, Blincoe K, Dietrich J (2020) Technical Lag of Dependencies in Major Package Managers. In: 2020 27th Asia-Pacific Software Engineering Conference (APSEC). pp 228–237. <https://doi.org/10.1109/APSEC51365.2020.00031>
- The Apache Software Foundation (2023) Apache Maven Project. <https://maven.apache.org/>
- Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Wu Y, Liu Y (2020) An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In: International Conference on Software Maintenance and Evolution (ICSME '20). IEEE, 35–45. <https://doi.org/10.1109/ICSME46990.2020.00014>
- Wonnacott TH, Wonnacott RJ (1991) *Introductory Statistics*
- Xavier L, Brito A, Hora A, Valente MT (2017) Historical and impact analysis of API breaking changes: A large-scale study. In: 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17). IEEE, pp 138–147. <https://doi.org/10.1109/SANER.2017.7884616>
- Xia B, Bi T, Xing Z, Lu Q, Zhu L (2023) An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. In: Proceedings of the 45th International Conference on Software Engineering (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, pp 2630–2642. 9781665457019. <https://doi.org/10.1109/ICSE48619.2023.00219>
- Zach (2021) What is Inter-rater Reliability. <https://www.statology.org/inter-rater-reliability/>
- Zhang L, Liu C, Xu Z, Chen S, Fan L, Chen B, Liu Y (2022) Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing. In: IEEE/ACM International Conference on Automated Software Engineering (ASE '22). ACM. 9781450394758. <https://doi.org/10.1145/3551349.3556956>

Authors and Affiliations

Dhanushka Jayasuriya¹  · **Samuel Ou¹** · **Saakshi Hegde¹** · **Valerio Terragni¹** · **Jens Dietrich²** · **Kelly Blincoe¹**

✉ Dhanushka Jayasuriya
djay392@aucklanduni.ac.nz

Samuel Ou
sou323@aucklanduni.ac.nz

Saakshi Hegde
sheg158@aucklanduni.ac.nz

Valerio Terragni
v.terragni@auckland.ac.nz

Jens Dietrich
jens.dietrich@vuw.ac.nz

Kelly Blincoe
k.blincoe@auckland.ac.nz

¹ University of Auckland, Auckland, New Zealand

² Victoria University of Wellington, Wellington, New Zealand