Dead Links and Lost Code: Investigating the State of Source Code Repositories in Maven Central Repository Packages

An Empirical Study

Tudor-Gabriel Velican[1]

Supervisor(s): Sebastian Proksch[1]**, Mehdi Keshani**[1]

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

Abstract

Maven Central serves as the de-facto repository for distributing free and open-source Java libraries and components. Evaluating its present state and overall robustness is pivotal for enabling the community to make well-informed decisions concerning its future progression. Such informed decisions would undoubtedly benefit the collective community of developers. This study aims to empirically evaluate developer practices surrounding version control and package reproducibility on Maven Central by investigating (i) the reliability of repository links, (ii) preferences regarding repository hosting services, (iii) the utilization of tags/releases, and (iv) the reproducibility of packages. Our study revealed that 20.85% of packages had unreliable repository links, attributable to inconsistencies in field usage and missing data, highlighting lax submission guidelines. GitHub emerged as the dominant host, with a market share exceeding 90% most years, though regional alternatives, like Gitee, are gaining traction. 74.35% of packages used tags/releases; however, naming convention discrepancies between Maven Central and source code repositories were identified, hindering version tracing and reproducibility. Strikingly, only a 3.06% of packages were configured to attempt reproducibility. An even smaller subset was found to be fully reproducible.

1 Introduction

Open-Source Software repositories play a critical role in the software development ecosystem. They are treasure troves of libraries and tools that facilitate the development of new software. One such repository that has gained popularity, particularly in the Java community, is the Maven Central Repository. Maven is a widely used project management and build automation tool for Java and other JVM-compatible projects, and the Maven Central Repository is a comprehensive database of libraries and plugins for these projects. While Maven centralizes and manages the binaries of these libraries, their source code often resides in different source code repositories like Github, Gitlab, and Bitbucket. As software development is an ever-evolving field, the state and reliability of these libraries are always in flux.

One crucial aspect of software repositories is the reliability of source code repository links. A broken repository link can hamper the development process, making it difficult for client developers to access needed resources, receive support, or contribute to the libraries. This has the potential to cause developers to abandon projects, which hurts the ecosystem as a whole [1]. When the pool of developers stagnates or shrinks, the ecosystem suffers from a lack of new perspectives and innovations. Furthermore, for developers and organizations to commit to integrating a library into their projects, there must be a level of trust in its stability and longevity. Having the guarantee that a library is in active development, with an engaged community and reliable access to its source code, bolsters confidence in its future sustainability and the feasibility of incorporating it into long-term projects [2] .

The location where the source code is hosted can have implications on its availability and reliability. With the dynamic nature of the web, hosting services may go in and out of operation, and repositories may move, raising the question of how this dynamism affects the Maven ecosystem.

Moreover, when a developer uses a library, they expect the the code to be stable. However, with constant commits and releases, bugs and breaking changes are inevitable. Pinpointing the commit associated with a release can help in recreating issues or understanding the state of the code at that release.

Another aspect to consider is the reproducibility of the packages. Reproducibility refers to the ability to reconstruct a package to the same specifications as the original. In other words, given the same source code, build environment, and configuration, the build process should yield the same artifacts bit-for-bit. This is an attribute of paramount importance from a security standpoint. It allows the integrity of packages uploaded to Maven Central to be verified. By ensuring that the build process always produces the same binary, developers can ensure that no unauthorized modifications were made to the source code [3]. This is especially relevant in enterprise environments, which often have strict acceptance criteria for third-party libraries and/or are subject to regulatory compliance.

It is worth noting that while some packages upload source JARs alongside the compiled binaries, not all do. Source JARs contain the raw source code from which the binaries are built. When they are made available, developers and security analysts can check the code for any vulnerabilities or malicious content and then compare it to the compiled binaries. However, in cases where source JARs are not provided, it becomes significantly more challenging to conduct a thorough security analysis. Moreover, there are no strict mechanisms in place to ensure that the source JARs directly correspond to the compiled JARs on Maven Central [1].

This underscores the importance of reproducibility. If a package is reproducible, its binaries can be built from the source code and directly compared to the binaries obtained from Maven Central, ensuring they are identical and have not been altered.

Progress towards ensuring the reproducibility of packages on Maven Central has been made thanks to the Reproducible Builds effort[2] and its contributions to the Maven build tool and its plugins. As part of this effort, the Reproducible Central project[3] exists to list all package releases that have been independently verified to be reproducible. To streamline the verification process, they offer scripts that facilitate the building and comparison of artifacts against their corresponding versions published on Maven Central. We leverage these scripts in our experiments. Although these advancements have enabled developers to create reproducible packages, it remains an optional feature.

This research aims to address the aforementioned critical aspects and answer the following research questions:

RQ1: How reliable are the repository links?

RQ2: Where are the repositories hosted and how does this change over time in the ecosystem?

RQ3: Can the commit pertaining to a specific release be pinpointed?

RQ4: How reproducible are the packages? Can one rebuild the packages with the same checksum?

[1]https://central.sonatype.org/publish/requirements/#supply-javadoc-and-sources

[2]https://reproducible-builds.org/docs/jvm/

[3]https://github.com/jvm-repo-rebuild/reproducible-central

First, for RQ1, we test the dependability of repository links in Maven's POM files using Git's `ls-remote` tool to check the validity and accessibility of repository URLs. Approximately 20.85% of packages lack valid Git repository links, and there is inconsistency in the use of fields to provide project URLs. This raises questions about the strictness of submission guidelines on Maven Central, as a small percentage of packages have missing mandatory fields.

Then we conduct a historical analysis for RQ2 by extracting hostnames from URL fields, aggregating this data to analyze the market share of various repository hosting services over time. Github is the most popular repository host, with over 90% market share for most years, though smaller hosts like Gitee and Apache's Gitbox still exist.

In RQ3, we examine tags and releases within Github source code repositories to determine the exact commit associated with a specific release. To match package versions with tags and releases, we query the Github API. The analysis also revealed naming convention discrepancies between the published Maven Central version and the source code repository, which can create difficulties for developers in finding the correct versions and ensuring reproducibility.

Lastly, for RQ4, we assess the reproducibility of packages in Maven Central. This involves verifying the presence of the *project.build.outputTimestamp* property, excluding packages without it, and employing scripts from the Reproducible Central project to build packages and compare their checksums. Reproducibility is a major concern, as only 3.06% of packages have the necessary configuration for reproducibility. This indicates a lack of knowledge or consideration among developers. Out of the packages we were able to automatically build, only 16% achieved full reproducibility, which is alarming in contexts where security and transparency are crucial.

The rest of the paper is structured as follows: Section 2 discusses related work in this field, highlighting important contributions made by previous studies and establishing the need for this research. Section 3 details the high-level approach and implementation used to conduct this research as well as the sampling strategy used. Finally, our findings and implications are discussed in Sections 4 & 5, respectively.

2 Related Work

The Maven repository has been the object of scientific inquiry for numerous researchers in the field of software engineering research. In this section, we discuss key relevant literature and how it pertains to our research goals.

Raemaekers et al. [4] have analysed the Maven Repository in detail, producing the Maven Dependency Dataset. Performed on a dataset of 148,253 jar files, code metrics, dependencies and breaking changes between library versions were gathered. Moreover, a call graph of the entire Maven repository was generated. The ultimate purpose of the research was to create a foundational dataset to allow other researchers to answer software evolution-related research questions on the Maven Repository.

In contrast, part of our research focuses on determining the reliability of source code repository links, providing insights into how well Maven packages are maintained. Furthermore, our research on whether individual releases can be consistently rebuilt from the remote source history could improve the accuracy of future research. For example, since the study was dependent on source code analysis, projects without a source jar directly available had to be discarded. If it were possible to rebuild source jars directly from source, it would certainly increase the reliability of such research.

Karakoidas et al. [5] performed a similar analysis (albeit on a smaller dataset of 22730 jars) which focused on collecting in-depth code metrics related to object-oriented design, package design and program size. They analysed packages using 3 popular static analysis tools, making available a dataset with the results. Using this dataset, they measured the usage of domain-specific languages such as Regex and XML. However, the number of other research questions that can be answered with the dataset is large. Yet again, the research was limited by the fact that packages that did not also contain the source jar were discarded, highlighting the importance of our research.

Tufano et al. [6] investigated uncompilable snapshots in the commit history of 100 Apache ecosystem projects written in Java and relying on Maven. They found that uncompilable snapshots occurred in 96% of the projects, with the main culprit being dependency resolution problems.

Similarly, He et al. [7] sought uncompilable commits in 68 open-source Java repositories to find a link between commit purpose and compilation errors. They also investigated their impact on subsequent commits and code quality thereafter.

Whilst our research also investigates compilability, we do this only at release level granularity, as our aim is to determine whether distinct artifacts can be rebuilt from the source code provided in the source code repository. Moreover, we try to associate artifacts with their respective commit by analyzing their checksum.

Whereas to the best of our knowledge there is no research on dead/non-existent repository links in the Maven Repository, there is research such as [8] by Liu et al. which investigated the prevalence of dead external links on Stack Overflow. At the time of writing, they found that 14.2% of external links in posts were broken.

3 Methodology

In 3.1, we discuss the sampling method used and provide a description of the dataset analysed in the study. Then, in 3.2, we describe the high-level approach employed to answer each RQ. Lastly, in 3.3 we delve into the implementation details, explaining the most important experiment design decisions.

3.1 Data Selection

As indexed by mvnrepository.com[4], the Maven Repository ecosystem contains upwards of 34M indexed packages across 1914 repositories. Organizations and individuals may host their own repositories and popular packages are often mirrored across multiple repositories for redundancy and availability.

We analysed a subset of packages, employing a data selection strategy formulated based on the sampling process suggested by H. Taderhoost in [9].

Sampling Frame

The largest and most diverse repository is Maven Central, with 11M packages. It is the default repository used by Maven and is highly regarded and trusted by the Java community. Moreover, it offers well structured and easily accessible metadata about packages in the form of a weekly-updated index. As such, we restricted our sampling frame to packages within Maven Central.

Sampling Technique

We analysed every package present on the Maven Central index available to date, performing a random sample to select

[4]https://mvnrepository.com/

a random version from each. The latest version was not selected as the sample would not be representative of each year. This is because packages from active projects started a long time ago would skew the dataset, leading to a lower representation from earlier years. To ensure reproducibility, we use a configurable seed for the random version selection.

Figure 1 shows the distribution of published packages per year. As can be seen by the skewness of the distribution, the popularity of the Maven repository has picked up in the last few years.

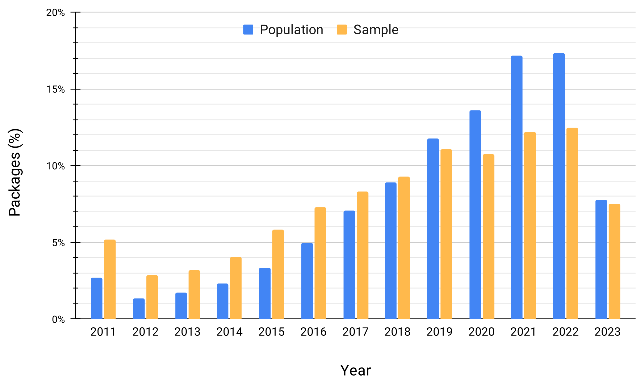

Figure 1: Distribution of packages published per year (population in blue, sample in yellow)

3.2 Approach

RQ 1 - How reliable are the repository links?

There are multiple fields within which developers could declare the URL of a package's source code repository inside the POM, namely:

- **package.url** - the home page of the project. Projects that do not have a homepage often include the repository URL here.
- **package.scm.url** - the source code repository, which should be a publicly browsable repository.
- **package.scm.connection** - used by the Maven SCM plugin to allow interfacing with the source code repository directly through the Maven command line. This URL should provide read access to the repository.
- **package.scm.developerConnection** - similar to the preceding item, except this URL should provide write access to the repository.

Note: when mentioning these fields in the rest of the paper, we will omit the *package.* prefix.

Although developers should declare the URL within the *scm.url* and/or *url* fields, Maven does not enforce these rules. They can declare the url in some, all, or none of the fields. Furthermore, according to the Maven documentation[5], *scm.connection* and *scm.developerConnection* should be declared in the following format:
{`scm:<provider id><delimiter><provider-specific part>`}.
Examples:

- scm:git:https://gitbox.apache.org/repos/asf/maven.git
- scm:git:git@github.com:akka/akka.git

[5]https://maven.apache.org/scm/scm-url-format.html

However, this is also not enforced and in practice we found that not all developers complied.
To validate the repository links we proceed as follows:

Considering the widespread usage of Git as the predominant version control system [10], leverage the `ls-remote` command provided by Git[6]. It allows us to view branches and references in a remote repository without having to clone or fetch from it. Consequently, if the command execution is successful, it can be inferred that the repository link in question is both valid and publicly accessible. We validate each field in turn and record whether the URL is a valid repository. Any packages that do not pass this validation process are categorized as either belonging to alternative version control systems or deemed invalid.

As an alternative, one might consider executing an HTTP HEAD request as a means of validation. Nonetheless, it is important to recognize that while an HTTP HEAD request can confirm the link's validity, it does not inherently verify that the link leads to a public Git repository. As an illustration, a generic link such as *https://github.com* would be regarded as valid, even though it does not specifically point to a Git repository. Therefore this approach was not used.

RQ 2 - Where are the repositories hosted and how does this change over time in the ecosystem?

In order to analyse the market share of each repository host over time, we first extract the hostname of each URL field by parsing it. URLs that have been previously identified as invalid are excluded from this analysis as the focus is on repositories that are accessible to the public. Subsequently, the extracted hostnames are aggregated on an annual basis. For each year under consideration, we tally the number of repositories using a particular hosting service. To determine the market share of each hosting service, we calculate the proportion of repositories using each service relative to the total number of repositories for that year.

Given the diversity of repository hosting services available, it is practical to set a threshold for market share, such that hosts with a market share lower than 1% are grouped into an *Others* category.

RQ 3 - Can the commit pertaining to a specific release be pinpointed?

To pinpoint the exact commit pertaining to a certain version, we can look at the tags and releases in the source code repository and search for the ones that match the package version.

In Git, tags are specific points in the repository's history that are marked with a unique identifier. They provide a way for developers to label and reference significant points such as major versions and feature releases in the codebase. On the other hand, releases are a non-native feature provided by some source code repository hosts which encapsulate tags, including release notes and (optionally) pre-compiled binaries or installation packages.

We can retrieve them by querying the APIs provided by the repository hosting services. Since Github is overwhelmingly the most popular repository host, for this research question, we investigate only packages hosted there. Using Github's API, we search for and record tags/releases that match the version name of the package.

RQ 4 - How reproducible are the packages?

Verifying the reproducibility of packages on Maven Central is met with a multitude of challenges, the most critical of which being that the output of the Maven build tool is timestamp-dependent by default. As such, the checksum of the produced

[6]https://git-scm.com/docs/git-ls-remote.html

artifacts is automatically different across builds. This makes it impossible to compare the majority of published artifacts with their respective counterparts built from source.

However, reproducibility can be manually enabled in the POM by setting the *project.build.outputTimestamp* property to a desired timestamp. This instructs Maven plugins that generate files to use this timestamp, which generally is enough to ensure reproducibility. The Reproducible Builds project has fixed most sources of variability, however there still remain sources of variability that can result in a package not being reproducible, or only partially so:

- **Version ranges in POM** - dependencies declared with version ranges lead to unstable version resolution over time, depending on which versions are available at compile-time. For example, a version range such as [*1.5*,) could resolve to version *1.5* now, but then resolve to version *2.0* in the future.
- **Line endings** - Windows uses CRLF for line endings, whilst Unix uses LF. Build plugins that generate content will use the line endings of the compiling OS.
- **Major JDK version** - Bytecode produced by different major JDK versions is generally different. Even with source/target JDK version defined in the POM, the JDK version used to compile makes the difference.

To determine whether packages are reproducible, we first exclude all packages without the *project.build.outputTimestamp* property. Furthermore, we exclude packages that do not have a valid repository link and corresponding tag, as this information is needed to retrieve the source code. We then use the scripts provided by the Reproducible Central project to build the packages. The scripts rely on a custom .buildspec file to be defined for each package, which declares the required build environment. If the project already has this file for a particular package, we use it, considering it has been verified by the contributors of the project. Otherwise, we attempt to create the file ourselves using the extracted data for each package and default build parameters.

For each package we build 2 iterations, one with LF newlines and the other with CRLF newlines. On completion, we record whether the build was successful, saving the output of the compiler. In case of a successful build, we additionally record the filenames of both the reproducible and non-reproducible artifacts, which are produced by the Reproducible Central scripts. We attempt to build only with the Maven build tool.

3.3 Implementation

Compiling intermediate data

To analyse the sampled packages, we built a Java project that runs multiple extractors on them to create an intermediate dataset. We stored this data in a database to facilitate straightforward querying. We extracted the following fields:

- Source code repository urls from the POM;
- Java versions defined in MANIFEST.MF inside the JAR. Some build tools include the Java version used to compile the bytecode. For example, if building with Maven, the Maven Archiver Plugin will store this information in the *Build-Jdk-Spec* entry[7].

From this dataset, we perform additional computations to answer the individual RQs in turn. Initially, we parse and read the full Maven Central index, storing the list of all packages and corresponding indexed metadata in a table. Then, using the seed 0.5, we perform the data selection, storing the sampled packages in another table. For each package, we run each extractor sequentially, extracting data from the POM file, the executable, and other artifacts included in the package.

Further analysis

With the intermediate data compiled, we perform additional analysis within a Python project. We query the database and perform more computations and requests to external APIs, the results of which are also stored in the database.

Initially, for each package, we extract the URLs from the intermediate data into a new table, parsing and storing the hostname for each URL. Each URL is then fed into the `git ls-remote` command, which returns the refs contained within a repository. Though the refs are not relevant for the RQ, if the refs are retrieved successfully we can infer that the repository is accessible and public.

There is no enforced format for these URLs, thus we have to normalise them into a format that is accepted by Git. For example, we need to remove the *scm:git:* prefix from the URLs in the *scm.connection* & *scm.developerConnection* fields. Other URLs are using the SSH protocol, which sometimes requires the SSH key of the developer. Converting the URL to the 'https' version often fixes this. Lastly, some URLs include the */tree/{branch_name}* suffix commonly found in Github links to specify a certain branch. However the `git ls-remote` command does not recognize this path as a valid repository. As such we remove this suffix. We then make a request with each transformation of each URL and once it is verified to be working, we store the transformed version in the database.

To obtain the tags and releases, we use Github's GraphQL API. It is however possible that developers may use a slightly different versioning scheme in the source code repository. Table 1 shows different versioning combinations we encountered:

Version	Tag	Release
3.0.0	3.0.0	3.0.0
1.0.0.Final	v1.0.0	Release 1.0.0
1.2.1	package-name-1.2.1	Package Name (1.2.1)
2.4.6	Release/2.4.6	Version 2.4.6
1.0.0.Alpha17	1.0.0.Alpha17	n/a

Table 1: Different ways of versioning across package version, tag names and release names

As such, searching only for exact matches would result in many tags and releases being missed. To mitigate this, we perform a string search. For tags, the Github GraphQL API provides functionality to query for a specific tag .

However, there is no counterpart for releases. To perform the search, we first exclude all releases that do not have the version as a substring of the release name. Then we use the Ratcliff/Obershelp string-matching algorithm [11] to find the closest match. It works by finding the largest common group of characters between two strings and using it as an anchor. It then examines substrings on the left and right of the anchor, calculating a score based on the number of characters found in common.

This approach allows for long release names such as *package-name-1.0.2* to be matched with version *1.0.2* despite the small similarity ratio of 0.43, whilst also excluding releases such as *1.0.3* which, although has a large similarity

[7]https://maven.apache.org/shared/maven-archiver/

ratio of 0.8, is a completely different version.

When re-building the projects from source, to automatically create the .buildspec file, we use the information previously extracted, namely the Git repo, the Git tag, and the major JDK version used for compilation, which is found in the MANIFEST.MF file in the Build-Jdk-Spec entry. Moreover, we also define the type of newline used by the library publishers. There are only 2 types of newlines therefore we attempt to build with each.

4 Results

This section presents the findings of our empirical study in a systematic and comprehensive manner. The primary outcomes include quantifiable measures and observations from our experiments, which directly align with our research questions.

RQ 1 - How reliable are the repository links?

Out of the 473,352 packages investigated, 97.45% of packages defined a repository URL in the designated *scm.url* field. Overall, 80.28% had at least one valid repository link. Out of the rest that did not have any valid repository links, 6.87% packages had unparseable links and thus could not be validated. This includes examples such as:

- Placeholders
  - PROJECT_URL
  - scm:git:{git}
  - 'https://github.com/' + user + '/' + name
- Malformed links
  - $https://github.com/kondaurov-scala/json.$git
  - >https://github.com/ashr123/exceptional-functions
- Junk
  - hikvision-artemis-sdk
  - none
- Empty strings ('')

However, not all developers used the *scm.url* field for its intended purpose. Instead of including a URL to the source code repository, they included a URL to the project's homepage, which instead should be defined in the *url* field. Vice-versa, there were developers who defined the source code repository URL in both tags or only in the *url* field, which is designated for the project's homepage. Some developers provided the SSH URL to the repository, which sometimes would require authentication, although converting the link to an HTTPS URL would make it accessible. We performed this conversion automatically.

In Table 2, we provide the percentage of packages containing each field along with the percentage of packages that contain a valid repository link in each.

Metric	Percentage
Has $\geq$ 1 URL field	98.70%
Has $\geq$ 1 valid URL field	79.15%
Has url	97.45%
Has valid url	40.36%
Has scm.url	97.45%
Has valid scm.url	75.46%
Has scm.connection	94.99%
Has valid scm.connection	69.76%
Has scm.developerConnection	79.41%
Has valid scm.developerConnection	49.52%

Table 2: Percentages of URL field usage and valid URLs for each

RQ 2 - Where are the repositories hosted and how does this change over time in the ecosystem?

We analyzed the market share of each repository host for every year starting from 2011, which is when the oldest package in the index was published. Only the hosts of packages with valid and publicly accessible repository links were considered. The market share for each URL field is presented separately because the repository host may differ across the fields.

Figure 2 shows the market share of all source code repositories over time for each URL field. Note that repository hosts with less than 1% of market share in a certain year were merged into the *Other* category. Our findings show Github emerging as the dominant source code repository, exhibiting overwhelming popularity with a market share exceeding 90% most years. Nonetheless, a noticeable trend reveals a gradual increase in the popularity of alternative repository hosts in recent years, with *gitee.com*, *git-wip-us.apache.org* and *gitbox.apache.org* being the most popular.

Interestingly, we were able to observe Apache's relocation from the *git-wip-us.apache.org* domain to the *gitbox.apache.org* in late 2018[8], with a clear reversion starting in 2018.

RQ 3 - Can the commit pertaining to a specific release be pinpointed?

Out of the 360,086 packages with a verified publicly accessible Github repository, 74.35% had a release/tag that was found to be a probable match according to our heuristic search algorithm and Github's API explained in 3.3. Figure 3 shows the proportion of packages with a matching tag and/or release.

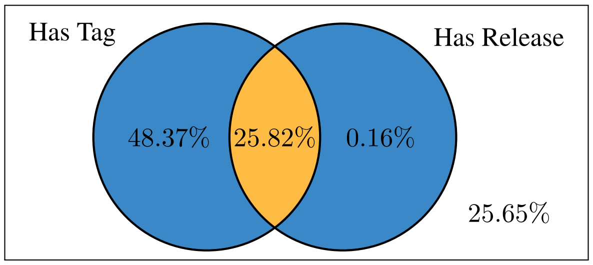

Figure 3: Venn diagram with percentages of packages having tags and/or releases

Taking the inherent imprecision of the search algorithm out of the equation, we also investigate the prevalence of tags/releases that exactly match the version name. The correctness of these tags/releases can be guaranteed with very high likelihood. Figure 4 shows the proportion of packages with a tag/release that has the exact same name as the package version. This time, only 24.45% were found to have a tag/release.

[8]https://infra.apache.org/blog/relocation-of-apache-git-repositories

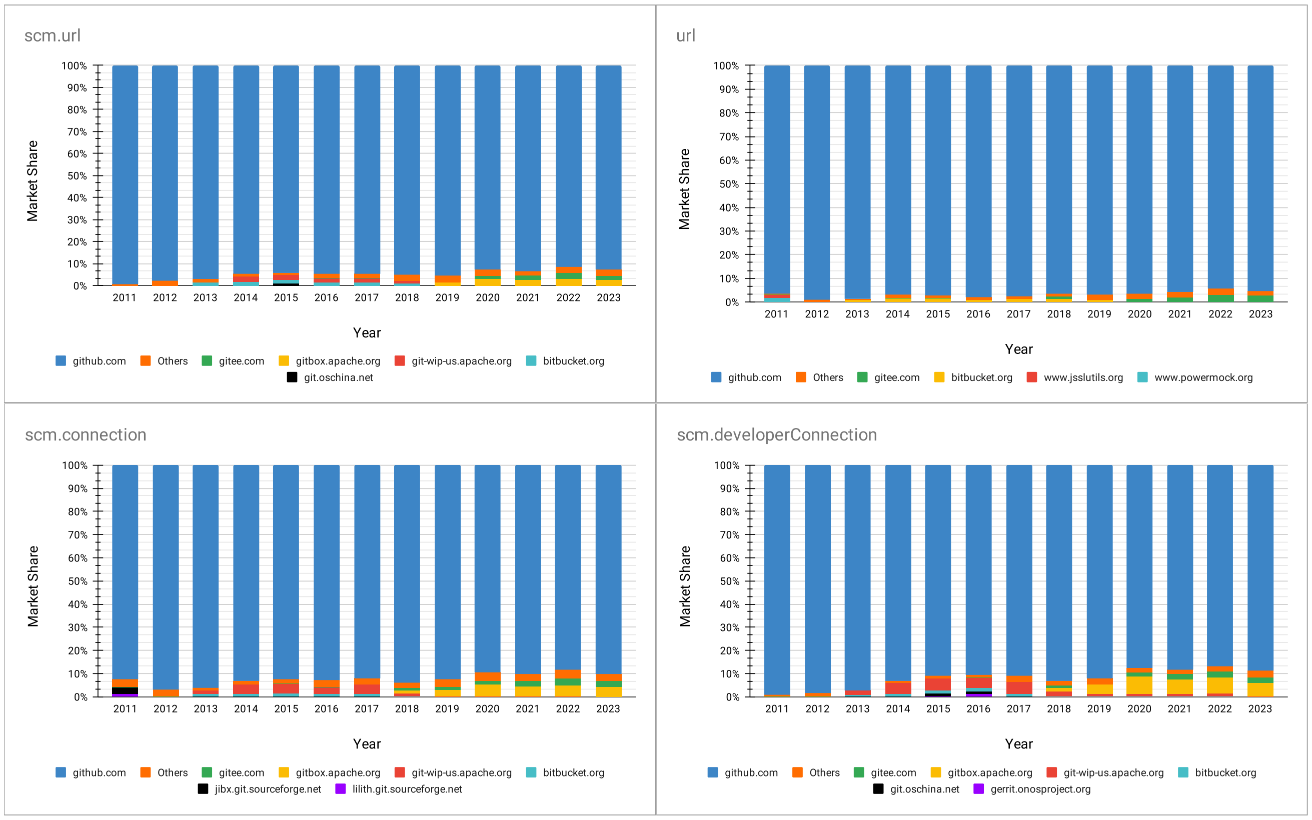

Figure 2: Market share of repository hosts per year for each URL field

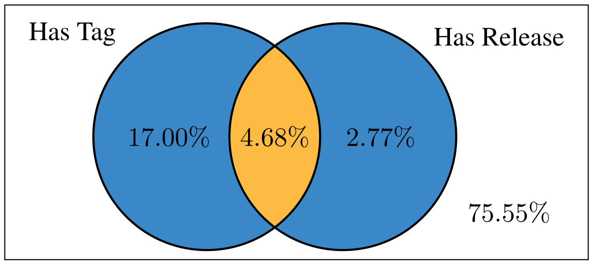

Figure 4: Venn diagram with percentages of packages having tags and/or releases whose names exactly match the version name

RQ 4 - How reproducible are the packages?

In fact, we found that only 9603 of the packages included the *project.build.outputTimestamp* property which is necessary to ensure reproducibility for projects built with Maven. In our sample, the first package found to contain the property was released in 2017, which is the same year in which the issue of reproducibility was opened in the Maven project's Jira[9]. 313,599 packages in the sample were published after the issue was opened on 25/Aug/2017. Out of this subset, 78,358 were released after 18/Apr/2022, which is when the issue of reproducibility was marked as fully resolved. However, we found many packages had started adding this property well before this date, which we attribute to the fact that work on reproducibility has been progressively rolled out over the years. Table 3 shows the number of packages containing this property since 2017.

[9]https://issues.apache.org/jira/browse/MNG-6276

Year (20xx)	17	18	19	20	21	22	23
Packages	1	0	2	954	2249	3674	2723

Table 3: Number of packages containing build.outputTimestamp property each year since 2017

Out of the 9603 packages, we attempted to build a subset of 481. Out of said packages, 8 had already been identified by the Reproducible Central project, enabling us to utilize their .buildspec files to set up the correct build environment. Surprisingly, we encountered 1 package that proved to be unbuildable, despite the verification of its .buildspec file by the project's contributors.

For packages not already identified, we automatically generated the .buildspec file using the data extracted previously. Only 230 could be automatically built successfully. Build failures often occurred because the builds required extra configuration that was not deducible without manually inspecting the documentation. Moreover, since we only built using Maven, packages using alternative build tools like Gradle or ANT failed. 37 packages were completely reproducible, 191 were only partially reproducible and 2 packages were not reproducible at all. The distribution of file extensions for the reproducible files is illustrated in Figure 5, while Figure 6 presents the corresponding distribution for the non-reproducible files.

Maven projects include a variety of file types, with POM and JAR files being the most common. Interestingly, while our results show that POM files are more common among reproducible files, JAR files have a higher presence among non-

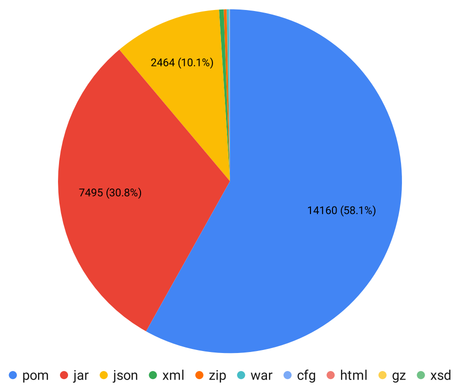

Figure 5: Pie chart with file extensions of reproducible files

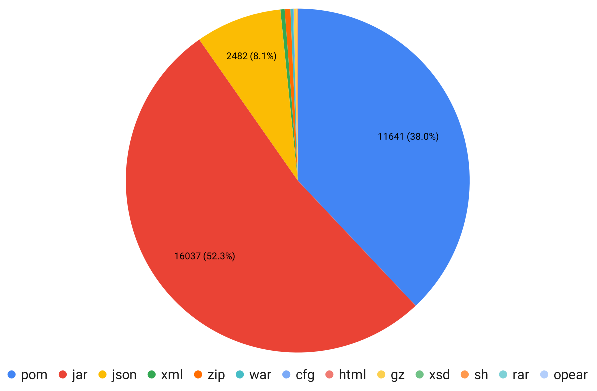

Figure 6: Pie chart with file extensions of non-reproducible files

reproducible files, hinting at possible challenges in ensuring reproducibility for Java archives. JSON files also appear to be common, although there is no significant difference in the proportion of reproducible and non-reproducible JSON files.

5 Discussion

5.1 Implications

Reliability of Repository Links We found that while the majority of packages include a URL in at least one of the designated fields in the POM, indicating that developers are generally providing some form of repository link, there is a significant portion of packages (approximately 20.85%) that do not have a valid URL pointing to a Git repository. Note, however, that some of the invalid links can be attributed to the fact that some packages use alternative version control systems such as Subversion, which were not explored in this research. This finding raises some concerns about the reliability of repository links on Maven Central, which can have serious effects on the ecosystem as a whole. Without a valid source code repository link, it is impossible to check the integrity of a certain package, as it is not possible to reconstruct it from source. Moreover, it can deter developers from using it, as they cannot assess its development practices, community support, or ongoing maintenance efforts.

We also uncovered a lack of consistency and understanding among developers regarding the appropriate usage of the *url* and *scm.url* fields. Developers often use them interchangeably despite the fact that *url* should be used for the project's website and *scm.url* for the project's source code repository. However, since specifying both is a hard requirement for submission[10], it is understandable that small projects without a dedicated website choose to specify their repository URLs in both.

Though a very small minority, the presence of packages with missing *url* and *scm.url* fields is surprising, despite it being a hard requirement for submission. This brings into question how thoroughly packages are reviewed when submitted to Maven Central. These findings highlight the need for improved enforcement of submission guidelines. Maven Central could also consider implementing mechanisms to validate and verify the repository links provided by developers, further enhancing their reliability.

Repository Host Preferences In our results, Github emerged as the dominant source code repository, exhibiting overwhelming popularity. It has maintained a consistent majority market share throughout the years, indicating its widespread adoption and recognition within the developer community. A notable finding however, is that while Github retains the lion's share, an array of smaller repository hosts continue to exist, catering to niche needs and specific communities. The Apache Git repositories, for instance, are mainly dedicated to projects under the Apache Software Foundation. This suggests that certain organizations may prefer to host their repositories in-house or on specialized platforms.

In recent years, *gitee.com*, a China-based repository host appears to be rising in popularity. This trend could be attributed to local preferences or regulatory policies in some countries favoring domestic services. The increasing share of *gitee.com* suggests the existence of a broader phenomenon of regional influence on repository hosting preferences.

Tag/Release Usage The fact that a substantial number of packages have tags indicates that tagging is a common practice, possibly because it's a simple and integrated way to mark specific points directly within the Git repository. We did however find that 0.16% had a matching release but no matching tag. Since a release encapsulates a certain tag, this indicates that whilst extremely rare, some developers use vastly different naming conventions only for their tags.

This poses a risk for developers who rely on these packages, as the discrepancy between version names and tags might cause confusion or difficulty in tracking the right version for their needs. Additionally, the absence of tags/releases in the repositories makes investigating reproducibility impractical, as the exact commit to build from is not known.

The data also suggests different levels of package maintenance rigor. Those with both tags and releases or with exact matches might be following a more structured release process, while others might be less formal. The relationship with other project maintenance metrics could be investigated in future work.

Reproducibility We found that a very small fraction of packages (9603/313,599) contained the *project.build.outputTimestamp* property, which is necessary for a Maven-built package to be reproducible. It is to be noted that all features enabling reproducibility have only been fully completed on April 18th 2022. In our sample, 78,358 packages were released after this date, with only 5489 having the property. This low adoption rate suggests that many developers are not giving enough consideration to reproducibility or might not be aware of the correct configuration options. It remains to be seen whether adoption

[10]https://central.sonatype.org/publish/requirements/#project-name-description-and-url

increases in the next few years. Nonetheless, this is a concern for the safety and transparency of the ecosystem. As such, it is important to bolster awareness among developers and increase the accessibility of documentation regarding reproducibility.

The fact that 230/481 packages were buildable raises concerns regarding the buildability of packages. This suggests that there may be other factors affecting the buildability which warrants further investigation. However, this may be the result of attempting to automatically build the packages. Large complicated projects are often split up into smaller packages contained within the same repository. As such, they require special build parameters, such as defining the subdirectory within said package is located. Such requirements can only be inferred from a project's documentation (if it exists). A potential fix for this issue would be to for the maintainers of Maven Central to follow in the steps of Debian, by creating a standardised .buildinfo file[11] which would have to be included as a hard requirement for submission on Maven Central. This file, which would contain the build environment and other relevant parameters could ensure that projects are always buildable from source, with information directly available on Maven Central.

Of the built packages, only 37 were fully reproducible. With such a low rate of reproducibility, it is clear that even when packages are buildable and have the required property, there remain challenges in ensuring that they are fully reproducible. This is significant, since partial reproducibility may simply not be enough in contexts where security and transparency is paramount. Further research is required to identify the specific barriers and challenges to achieving full reproducibility, although the Reproducibility Central project has laid the foundations and continues to make contributions towards this issue.

Lastly, the distribution of file extensions shows that Maven projects are composed mainly of POM and JAR files. The large amount of non-reproducible JAR files found is concerning, as they are executable and are thus a risk to run without access to the source code [3].

5.2 Threats to Validity

External validity concerns the generalisability of our results. In this study, the main threat to external validity is that the Maven Repository ecosystem is larger than just Maven Central. As such, the conclusions drawn from our study may not be generalised to other repositories, which may have different rules and conventions for submission, though this can be investigated in future work. Moreover, there are also many private repositories used within companies which are not accessible.

Internal validity refers to the factors that could potentially impact the outcomes of our study. For example, although the sample contains every package in Maven Central, the version chosen for each package is random. Over time, although unlikely, the source code repository may be migrated or become deprecated. This means that older version of packages may not have valid URLs. In future research, it could be interesting to investigate the extent to which repository hosts change over a project's lifetime.

Another such factor is that the reliability of packages using other Version Control Systems such as Subversion and Mercurial is not verified.

Lastly, the prevalence of corner cases may have an impact on the number of packages marked as valid. For example, due to strange formatting of URLs or small mistakes, the host of the URL cannot be parsed and verified, although a human would easily be able to fix such broken links.

[11]https://wiki.debian.org/ReproducibleBuilds/BuildinfoFiles

6 Responsible Research

When performing this study, we had to access and download information about a large number packages and source code repositories that is provided publicly and without charge. As such, we made sure to follow the terms of use of the online services we used to the best of our abilities.

To lower the amount of data downloaded from Maven Central, we configured the application to use the already cached local repository of our research group[12], thus only downloading missing packages. Moreover, the application works on a need-to-know basis, only downloading the files it actually analyses as opposed to all files listed under each package. To analyse the Github repositories of the packages, we used the Github API, which imposes rate limits. We followed these limits strictly, using a single API key and waiting for the timeout to reset before performing additional requests.

We made sure to make our results reproducible by preparing a replication package, publishing the codebase publicly on Github[13]. Within the replication package[14], we also provide the weekly Maven Central index used in the study.

7 Conclusion

Our research investigated the reliability of repository links, repository host preferences, tag/release usage in said repositories and the reproducibility of packages in Maven Central.

The dependability of repository links was discovered to be compromised, with around 20.85% of packages lacking a valid Git repository link. Furthermore, there was a lack of consistency in the use of fields to provide project URLs. Although the lack of valid links may be attributable to the usage of alternate version control systems or small projects without dedicated websites, the fact that a small percentage of packages had missing mandatory fields questions the strictness of submission guidelines on Maven Central. This highlights the necessity for stricter enforcement of submission guidelines and implementation of mechanisms to validate and verify repository links.

Although smaller repository hosts such as Gitee and Apache's Gitbox continue to exist, Github was shown to be the most popular platform in terms of repository host preferences with an overwhelming majority, exceeding 90% market share for most years.

While 74.35% of packages used tags or releases, our analysis also found that naming conventions varied between the published Maven central version and source code repository, which is concerning. Developers may encounter difficulties finding the correct versions and determining reproducibility as a result of this mismatch. We also found that certain developers may adhere to more structured release processes than others in terms of package maintenance rigor.

Most importantly, reproducibility emerged as a key concern. Only 3.06% of packages were found to have the configuration required to ensure reproducibility. The current situation shows a lack of knowledge or consideration on the part of developers. Further, even when packages were buildable, only 16% achieved full reproducibility. This is alarming, especially in contexts where security and transparency are critical.

[12]https://www.fasten-project.eu/

[13]https://github.com/ashkboos/MavenSecrets

[14]https://zenodo.org/record/8077125

References

[1] E. Constantinou and T. Mens, “Socio-technical evolution of the ruby ecosystem in github,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 34–44.

[2] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, “Categorizing developer information needs in software ecosystems,” in *Proceedings of the 2013 international workshop on ecosystem architectures*, 2013, pp. 1–5.

[3] C. Lamb and S. Zacchiroli, “Reproducible builds: Increasing the integrity of software supply chains,” *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022.

[4] S. Raemaekers, A. Van Deursen, and J. Visser, “The maven repository dataset of metrics, changes, and dependencies,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 221–224.

[5] V. Karakoidas, D. Mitropoulos, P. Louridas, G. Gousios, and D. Spinellis, “Generating the blueprints of the java ecosystem,” vol. 2015-August. IEEE Computer Society, 8 2015, pp. 510–513.

[6] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “There and back again: Can you compile that snapshot?” *Journal of Software: Evolution and Process*, vol. 29, 4 2017.

[7] J. He, S. Min, K. Ogudu, M. Shoga, A. Polak, I. Fostiropoulos, B. Boehm, and P. Behnamghader, “The characteristics and impact of uncompilable code changes on software quality evolution.” Institute of Electrical and Electronics Engineers Inc., 12 2020, pp. 418–429.

[8] J. Liu, X. Xia, D. Lo, H. Zhang, Y. Zou, A. E. Hassan, and S. Li, “Broken external links on stack overflow,” *IEEE Transactions on Software Engineering*, vol. 48, pp. 3242–3267, 9 2022.

[9] H. Taherdoost, “Sampling methods in research methodology; how to choose a sampling technique for research,” p. 5, 2016. [Online]. Available: https://hal.science/hal-02546796

[10] J. C. Casquina and L. Montecchi, “A proposal for organizing source code variability in the git version control system,” in *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 82–88. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3461001.3471141

[11] J. W. Ratcliff and D. E. Metzener, “Pattern-matching-the gestalt approach,” *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.