Effects of Artifact Age on Maven Dependency Resolution

Gints Kuļikovskis[1,⋆]

Supervisor: Sebastian Proksch[1]

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

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 29, 2024

Name of the student: Gints Kuļikovskis⋆
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Casper Poulsen

*

Abstract

This study conducts an investigation of the challenges faced by aging projects in Maven Central, focusing on the issue of missing dependencies. Using the Maven Explorer indexer, we systematically examine the correlation between the age of a project and the frequency of dependency resolution failures. Our analysis reveals a notable trend: older packages in Maven Central are more likely to encounter dependency resolution issues compared to newer ones. A widespread cause that was identified is the reliance on repositories without Transport Layer Security (TLS). Through this research, we highlight the prevalent issues within the Maven Central ecosystem and also offer insights into common causes of dependency resolution failures. We advocate for uploading new versions of libraries to multiple repositories to mitigate these issues. This study reviews the current state of Maven Central and extends some of the findings to other package management systems, contributing to a broader discourse on software longevity and dependency management.

Keywords: Maven Central, Dependency resolution, Software longevity

1 Introduction

This research is motivated by the need to understand to what extent aging projects in *Maven Central* become unusable due to missing dependencies. While Maven Central itself preserves all past versions of artifacts, the artifacts themselves often have dependencies that are either located on alternative repositories or file hosting services, or are linked as individual files on the developer's machine. In fact, there have even been cases where such alternative repositories have been completely taken down, as in the case of JCenter [1]. Consequently, a user of some Maven project that has not been updated in a while might find that it is impossible to build the project, even though nothing has changed in the project itself.

This issue is particularly significant for open-source projects. A small but popular project that goes missing can cause worldwide disruption, such as in the case of the package "kik" from 2016 [2]. Similarly, researchers trying to replicate a previous experiment that now has a missing dependency may find themselves unable to do so [3]. This problem extends to researchers looking into historical trends of the Maven ecosystem as well, for example, because they cannot reproduce the results of their predecessors.

We build upon an existing indexer, Maven Explorer [4] in order to answer the question of whether dependencies become unavailable as a project ages and, for those that do, what are the most common causes. There exist more than 12 million distinct artifacts in Maven Central and at least 37 million global artifacts, with this figure representing artifacts that are indexed by [5] at the time of writing. We highlight stategies for having older releases remain resolvable for longer, in the context of Maven and Maven Central. We also question the relevance of our findings with respect to other package management systems such as Microsoft NuGet [6] or Debian's Advanced Packaging Tool (apt) [7] in Section 5.

The research topic is answered in two sub-questions:

1. What is the correlation between the age of artifacts on Maven Central and the frequency of their dependency resolution failures?
2. What are the predominant causes of dependency resolution failures in artifacts on Maven Central, and how do these causes vary by artifact age?

We present the results relevant to the research questions in Sections 3.1 and 3.2, with an explanation and further discussion in Section 5.

We have found that, in general, older dependencies are more often unresolvable: refer to Section 3. Furthermore, we investigate the proportion of dependency resolution errors related to now-blocked non-HTTPS repositories, estimating that close to 0.3 percent of all requested artifacts could fail to be downloaded because of this.

2 Methodology

In this section, we explain our overall approach for researching dependency resolution issues in aging packages as well as the approach for identifying some of the underlying causes. First, we explain the general steps, after which, in Section 2.1, we introduce some core definitions with respect to Apache Maven. After that, in Sections 2.2 and 2.2, we highlight the general data collection and processing steps used in this research. Lastly, we cover expected causes for dependency resolution failures.

This study utilized a quantitative research approach, focusing on indexing Maven Central's artifacts to assess how many of them have missing dependencies. We also checked the impact of having insecure HTTP-based repositories now automatically blocked by assessing whether the project references repositories with an address beginning with "HTTP://". Lastly, we gather information about the original transitive dependencies that are responsible for failing the dependency resolution step of packages in our data set.

2.1 Maven Repositories and Dependency Resolution

In this subsection, we introduce specific Maven concepts relevant to this research. The definitions are documented in more detail in [8], which is our main source for this subsection.

Maven dependencies are generally located in a repository that hosts information that is necessary to build and reference the package. In theory, any file that can be uniquely identified by its *group ID*, *artifact ID* and *version*, abbreviated as "GAV", constitutes a maven artifact. A developer can reference artifacts in their project by adding them to a file called POM.xml, providing the group and artifact identifiers and optionally the version of the desired package. Here, the desired scope of the dependency can be specified as well, for instance, to separate compile-time dependencies from those required in the testing stage.

When a project is built, the *local repository* is searched first to see if the required version of each artifact has already been downloaded. Otherwise, *Maven Central* and all explicitly declared repositories are searched for the artifact, until a match is found that fits the version requirements. The assembly of a project's dependencies is one of the first steps of a Maven build pipeline, executed using the 'dependency:resolve' task of the Maven executable [9]. Dependency resolution itself is a complicated process that often involves situations such as clashing dependency requirements, missing dependencies, and complex build lifecycles and plugin configurations. This research focuses on investigating the different errors that occur in dependency resolution; such errors generally interrupt the process itself and any subsequent build steps.

2.2 Data Collection

The original data source for this project was the metadata of a portion of Maven Central artifacts that was indexed over a set period of time using the [4] project. The collected data was recorded in a database and summarized using statistical methods to get an overview of the amount and types of dependency resolution errors. We looked at some of these categories to attempt to explain their general causes and, for some of those, suggested concrete mitigation steps.

Procedure

In order to answer the research questions, we rely on an index of the Maven Central repository. The repository is indexed using the Maven Explorer tool [4], which publishes its results as messages to separate Apache Kafka topics. Currently, around five million different releases are indexed, which is a number that is significant enough to develop an understanding of the overall state of the platform.

Apache Kafka is a platform for transporting messages from a producer to a consumer, with a focus on scalability and fault tolerance, as well as the ability to handle large volumes of data[10].

We implemented a Kafka consumer for processing the messages, storing the artifact's creation date, any encountered error messages, and useful metadata like the artifact's location, repository address, etc. This information is stored in a relation database and is later queried to compute statistics about the indexed artifacts. We then generate a histogram of the number of dependency resolution errors out of the total number of packages for each year.

The metadata was made available in Kafka Messages, which are categorized in three Topics follows:

1. Requested—Releases found in indices of Maven Central, to be downloaded;
2. **Downloaded**—Releases for which downloading was attempted, including the required dependencies; This topic was the focus of this research.
3. Analyzed—Releases for which additional processing was performed, such as the construction of an *effective POM*.

Each of the topics is divided into *Kafka Lanes*–Normal, Error, and Priority, based on the success of the previous task. The Priority lane was created to enable certain processing stages to re-enqueue artifacts with additional priority, skipping a line of potential thousands of other messages.

We analyze all available messages in the 'Downloaded' lane, merging Normal and Priority lanes into a "success" group, and investigate the "error" group separately. For calculating the proportion of artifacts with dependency resolution errors, we separated the artifact releases by year and divided the number of packages with errors by the total number of packages published that year.

Error Classification

The error messages consist of two parts: Exception class name and stack trace. We match the messages by the exception name to generate an overview of the most common errors that occur while retrieving the projects and their dependencies. We check the frequency of each type of exception in the set of packages that were attempted to be downloaded and manually inspect some examples from each of the broader exception classes.

Finding the Publishing Date

This research compares the age of the artifact with the presence of issues with dependency resolution. We considered two approaches to obtain the publication date. The most accurate way would be to execute a HEAD request to the HTTP resource that contains the analyzed artifact. An alternative and computationally lighter approach is to use the build date that is available with the indexed artifact. Although the build date might differ slightly from the actual publication date, we expected the difference to not exceed one or two months, in the worst case.

2.3 Potential Causes of Resolution Errors

Having approximated the general causes of a given category of errors, it is useful to investigate the underlying causes. In the scenario where the immediate error is an unsuccessful HTTP request, depending on the status code of the response, certain conclusions about the underlying cause could be drawn. Otherwise, the error source can be narrowed using the 'dependency:resolve' task of the Maven program, discussed in more detail in Section 3.

The HTTP 403 (Forbidden) status code is returned when the user is not authorized to access the requested resource. We encountered this issue in scenarios where the repository requires authentication, such as the GitHub Package Registry. In this case, the user can register their credentials in their Maven settings.xml, which will then be used to authenticate the request to the GitHub domain.

Repository Down

For a package with a URL pointing to a resource that is unavailable, it could be checked if the domain name can be resolved, or it could potentially be simply checked against a list of known former repositories that have been taken down. For example, while the JCenter repository was taken offline in 2018, in 2023 still more than 9.3 thousand pom.xml files on GitHub reference this former repository [11]. Note that a subset of these GitHub repositories is the source code of existing Maven artifacts.

Changes in the Surrounding Configuration

Migrations to the Maven system likely affected the availability of some artifacts. A conflict with the POM.xml schema is an unlikely but possible cause, since the schema has been developed to be backward compatible to some extent. More likely, incompatibilities with new maven plugins or other artifacts could cause issues after an upgrade of a Maven repository.

2.4 Exploring Solutions for Missing Dependencies

When trying to recover a missing artifact, several strategies emerge. A valid approach could be to use a mirror of the repository if the artifact's version is available there. However, it might be wise to verify the authenticity of the package. Checking for the GPG signature of the author can help, but this approach has its challenges, especially if the original package and the details of the original author are lost. This raises the possibility that the mirrored version's author could be different, with a distinct GPG key.

In situations where an older version of a required dependency is removed from the repository due to its age, newer versions might retain information about the author's GPG key. This detail can be useful for validating the integrity of the older version sourced from the mirror.

2.5 Manual Workarounds

Since some errors are the result of the resolver refusing to download dependencies from insecure HTTP repositories, potential workarounds were investigated and are covered in Section 3.

Furthermore, a trend was observed for POM.xml files to contain invalid xml tags (that is, those that conflict with its XML schema (POM.xml schema: [12]). In cases where this is the main cause of dependency resolution failure, these tags could be removed for parsing, further explained in Section 3.

3 Experimental Setup and Results

In this section, we provide specific steps that were made to answer the research questions,

The general setup of our experiment consists of a program called "Maven Error Stats" which parses the results of the Maven Explorer project, in the form of Kafka messages, and persists them in a relational database. The relational database is by default auto-generated based on our model classes, but can be reviewed in Appendix A.

Afterwards, we generate summaries using our repository containing Python scripts, called "Maven Error Summarizer". Here, we take advantage of our relational database, such as efficient storage and querying of rows; explained, for instance, in [13] among many others. These advantages are especially relevant in the context of generating an overview of a large number of rows. We therefore chose to build the project around executing most of the selecting logic inside our queries to the relational database, to use features like indexing and query optimization.

1. For packages that failed the dependency resolution task, we execute the task 'mvn dependency:resolve' and parse its output to find out exactly which dependencies are unavailable. With this functionality, it is possible to locate clusters of packages that all fail dependency resolution because of the same missing package. The results of this are available in Table 2
2. For all packages, we support locating the artifact in the local repository and persisting its POM file(s). This is useful because an error in the POM file of a package often causes the Maven dependency resolution plugin to fail. However, this was not used for this research.
3. To attain an approximate proportion of artifacts failing dependency resolution due to missing Transport Layer Security for the repository address, we evaluate the effective POM [14] of the artifact and search it for "<repository>" tags that contain an address attribute beginning with "http://".

Either of these additional processing tasks can be toggled using the appropriate flag in the system environment as explained in the README of our repository.

3.1 Research Question 1: Correlation Between Age and Resolution Errors

The Maven Explorer program is used to index packages found in the Maven Central repository. Maven Explorer caches the found packages in the local repository and publishes a Kafka message on the appropriate topic, 'normal' or 'error', depending on whether its dependencies could be resolved. These messages are then stored in a relational database system to enable querying on the collected information, including the release information, publishing date, and the type of error found while trying to resolve the dependencies of the package.

At the time of the experiment, our Maven Explorer instance has we used has saved 3.5 million versions from index number 700 starting on 2021-09-28, caching dependencies of projects released until January, 2024. The use of these indices therefore causes the releases published after 2021-09-28 to be overrepresented in our sample, as shown in Figure 2. In light of that, we exclude release dates after the first index, this way only showing packages that were stored because they are referenced by projects covered by the indices.

This research found that older packages are indeed more likely to fail dependency resolution, based on Figure 1, with the margin of error of 0.26% assuming a population size estimate by [5] of 37 million artifacts and 140'527 included in the experiment with a confidence level of 95%, calculated using [15]. Note that the estimate is a low estimate due to the fact that it only represents packages indexed by [5].

A numerical summary of the results from Figure 1 is as follows:

- Mean Proportion: 0.0312
- Median Proportion: 0.0248
- Standard Deviation: 0.0226
- Correlation Coefficient (R value): -0.838
- Linear Regression: $y = -0.004118x + 8.322981$

3.2 Research Question 2: Predominant Causes and Relation to Age

A number of different error messages were found when querying the database for artifacts whose dependency resolution has been unsuccessful. The most common general errors and their prevalence are illustrated in Figure 2 and Table 2. One of the

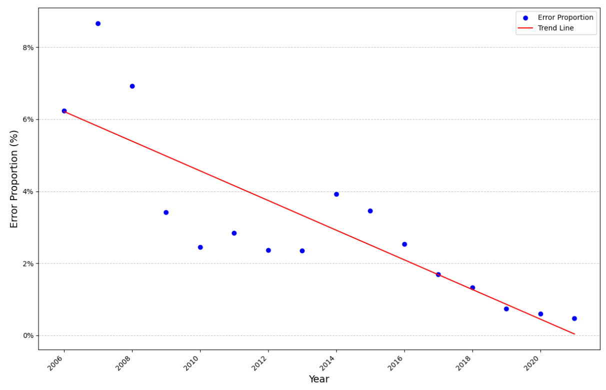

Figure 1: The proportion of artifacts with missing dependencies per release year. Data table available in Appendix C.

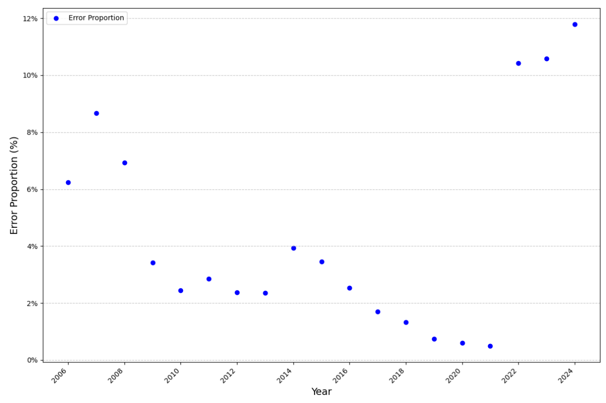

Figure 2: The proportion of artifacts with missing dependencies per release year, including over-respresentation of packages explicitly downloaded by the Maven Explorer (from 2021 onwards).

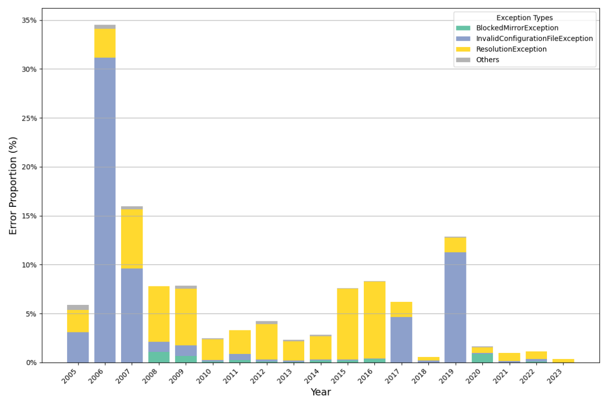

Figure 3: Proportion of different exception types encountered per artifact release year.

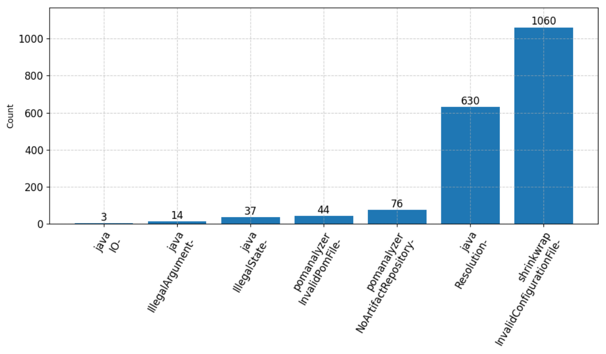

Figure 4: Most frequently encountered exception types in analysis pipeline.

most common errors occurs due to a security measure in place by default on the maven system to block repositories that do not support HTTPS, instead only providing HTTP-based access. While blocking unencrypted HTTP transport is a useful security measure to prevent various types of attacks or eavesdropping, it also prevents the dependency resolution of a significant amount of older artifact versions. Consider Figure 3 where the proportion of two most common error types plus the BlockerMirrorException, which indicates that dependency resolution failed because the transport layer security requirement was not met. Considerations and workarounds for this problem are discussed in Section 5.

By evaluating the effective POM [14] of the artifact and searching for "<repository>" tags that contain an address attribute beginning with "http://", we attempted to estimate how frequently artifact releases depend on repositories that do not support Transport Layer Security. We did not collect a significant amount of data points using this approach, estimating a share of 0.3% based on 1197 artifact releases.

In order to retrieve more detailed information about the source of errors in dependency resolution, the maven action 'mvn dependency:resolve' is executed with the given artifact as input. With this, it is possible to find the path to the specific dependency that cannot be retrieved, allowing for better insight into the underlying issue.

In Table 1, we list artifacts that are most often found missing when running 'mvn dependency:resolve'. It may be interesting to look into the specific cases of each artifact. Note that the number indicated by the first element, the Maven Downloader Plugin, is the first dependency that needs to be fetched when checking any artifact. Therefore, it is present in the output for all analyzed packages, consequently indicating the exact number of artifacts that were analyzed. These results are not indicative of an issue in the context of this research, instead showing an interesting correlation that could be investigated in future studies.

4 Responsible Research

This research investigates a topic that affects open research—when a repository containing a research experiment no longer compiles, it is no longer easily repeatable.

In this section, we cover the topics of data collection ethics, auditability of our source code, and potential threats to validity.

maven-downloader-plugin	org.apache.maven.plugins	40404
glide	com.github.bumptech.glide	962
annotation	androidx.annotation	943
appcompat	androidx.appcompat	938
hilt-android	com.google.dagger	807
wasp-model_2.11	it.agilelab	765
core-ktx	androidx.core	693
booster-android-gradle-compat	com.didiglobal.booster	638
network	me.proton.core	608
network-data	me.proton.core	600

Table 1: Top 10 artifacts that are most often missing, causing dependants to fail dependency resolution. A full version of this table in Appendix B.

4.1 Data Collection Ethics

The data that was collected as part of this research consists of metadata of publicly available Maven packages. Our exporter program supports cataloging the contents of files found in artifacts, but this functionality was not used for this research. Furthermore, this information was already available in the source repository, and our project does not make it easier to traverse it. We therefore consider the collected data not to be especially sensitive.

The database is available upon request.

4.2 Open Science

Every step of the analysis pipeline is open source and is also available as a self-contained docker image. We published the source code under the MIT license, so that anyone can reuse, adapt, and distribute it without restrictions.

The program created as part of this research is w with simplicity in mind, to prevent technical complexity from posing as a barrier for review efforts. The source code is available at https://gitlab.com/gintsk/maven-error-stats, and a Docker image is published at https://hub.docker.com/repository/docker/gints1/maven-error-stats.

The program depends on the Spring JPA Starter repository, which has some drawbacks, but greatly reduces the amount of repetitive code by auto-generating the database access layer. Some of the drawbacks include unnecessary transitive dependencies and essentially forcing to choose the Spring '@Bean' dependency injection instead of, for instance, Google Guice. Additionally, the relevant database connection libraries are updated as a single package, reducing the chances of future dependency conflicts.

The project that was created as part of this research was built to be extendable with new features. It is published as a self-contained docker image, for easy replication of the work.

A core decision that we made is to avoid extending third-party Java code, even if this requires additional work. In early development, we modified parts of the Maven Explorer project[4] and the Kafka wrapper project Franz[16], to make changes and add needed functionality. Later we switched to using Confluent's *parallel-consumer* library[17], for a simple multithreaded consumer support. This way we were able to remove the last remaining references to packages hosted on *GitHub Maven Package Registry*, consequently removing the need for user authentication when building our project or its docker image. As a result, the resulting JAR is smaller (50 MB instead of 220 MB) and the program takes only 15 seconds to compile instead of close to two minutes, informally tested on the author's machine (x86-64, 24-core i7-12800HX, 32GB RAM, 100Mb ISP link; command 'docker build .'). This makes it easier to test the self-contained image while making incremental changes. In the program, it is also possible to specify the exact configuration of Kafka, which is the event management system that we use. We also support changing the number of virtual threads, opening up the possibility to run the project reliably on resource-limited systems as well as vast server clusters that have a lot of resources to spare.

4.3 Repeatability and Generalizability

Our image was manually tested on multiple systems running the Maven Explorer on other systems, and the analysis was repeated on the dataset from a colleague's machine. Using our Docker image, it is easy to repeat our experiment on various different systems, and the results databases are easily merged using PostgreSQL pg_dump utility. We consider that our code repository opens the possibility to explore other research questions, using the same data set.

With regard to internal validity, we consider that, using the explanation in Section 3 for the experimental setup and the README.md of our repository for practical information, it is possible to replicate the experiment. We consider that there is at least one significant threat to internal validity, as in our experiment, TLS-lacking repositories were enabled half-way through the experiment. When collecting a new dataset to repeat our experiment, one would make a choice of whether to allow through insecure HTTP requests before starting the collection. This would result in a different proportion. Moreover, it is highly likely that a dataset collected by another researcher will contain different artifact releases, even if they are using the same Maven Central indices for choosing which artifacts to request.

About external validity, we consider that the problem covered by our research is relevant for other package management systems like Microsoft's NuGet and Debian's Advanced package tool (apt). When dependencies become unavailable, projects usually cannot be used without making changes to the source code or the surrounding configuration. Other package management systems also experience similar issues, for example, as NuGet's Documentation about insecure HTTP repositories[18] shows, this topic is relevant in their ecosystem as well. Debian's apt approaches this issue differently, not requiring transport layer security but instead checking that all packages are signed with a GPG key [19]; whereby this aspect of our research does not currently apply.

Although there have been existing studies like [20] comparing the properties of different package managers, we consider that it might be interesting to compare the results of this study with other package managers.

5 Discussion and Future Work

The findings show that older packages are more likely to fail dependency resolution than newer packages, with a downward trend towards the present. We explain this as breakage due to external changes that are made to the host repository or the Maven ecosystem such as repository migration or upgrades to surrounding configuration such as POM file structure. The problem of increased dependency resolution failures for older packages are likely further exacerbated by the use of external repositories, or other factors that we have not considered—which is worth looking into in future research.

The experiments conducted in this research do not indicate a connection between different dependency resolution errors and the release date of the artifact.

Although from an insufficiently large data set, we estimate that around 0.3% of artifacts could be located on repositories that are reached over HTTP without Transport Layer Security, further discussed in Section 5.3.

5.1 Bloated Dependencies

Numerous other projects have been created for analyzing the issues relating to the decentralized nature of Maven. [20] found that 75% of all declared dependencies across the Maven ecosystem are unnecessary. These dependencies are declared in the *pom.xml* file, and even though they are not used in the project, maven will still attempt to resolve them.

We think that this might be explained by the fact that developers are not always aware of the dependencies that are included in their projects. Dependency management could be considered a chore to some, in contrast to managing the actual program that interacts with the linked libraries. Reasonably, developers may forget to revisit this task unless problems arise, meaning that a dependency that had been added temporarily might not be removed before the project is published.

There also exist so-called "starter" dependencies, which allow the developer to start working with a certain framework without worrying about individual components [21]. These take away the task of updating every individual component while significantly increasing the resulting artifact size. We think that it is generally advantageous to use these dependencies as they reduce the risks of individual components being outdated or incompatible with each other.

5.2 Sources of Failure for Missing Dependencies

We recommend investigating individual artifacts that are most often found in the trace obtained when executing the dependency resolution task. This research does not cover this question, but we consider that this could provide useful insight into this research area.

5.3 The Lack of TLS in Maven Repositories

The use of a repository that lacks transport layer security (TLS) is the most common issue. For individual developers, if there is a need to use an older project that relies on dependencies hosted on an HTTP-based repository, it might seem appropriate to remove the HTTP-based repository blocker. However, this could introduce a security risk [22]. It is a much better idea to unblock individual repositories as needed, as explained in this StackOverflow answer: [23].

In light of the issues discussed, a general recommendation is to avoid using more than one repository in open-source projects. Especially for more popular open-source projects, we recommend uploading new versions of your library to multiple repositories.

'System' and 'Provided' Scopes
Maven allows declaring local dependencies using the scope 'system', specifying that the location of the package is a URL not inside a repository but the local system [12]. Maven dependency resolution fails if any path specified in this way is absent. Similarly, the scope 'provided' indicates that the jar is expected to be present in the environment, such as one of the Java system libraries. We did not investigate whether the use of these scopes is widespread; it may be worth investigating in future work.

5.4 GitHub Package Registry

Why do developers choose to publish packages in the GitHub package registry instead of Maven Central? One reason is that it is simply easier to do, rather than having to meet the requirements to publish a package on Maven Central [8]. [24] found that the GitHub package registry is the third most popular hosting repository for Java packages, highlighting that it is more popular for publishing than for referencing packages. We believe that this is reasonable, given that the use of packages from the GitHub package registry requires additional configuration. Based on our results, we generally discourage the publishing of open-source packages on package registries that require authentication, like the GitHub package registry.

5.5 Summary

In conclusion, this study provides insight into the challenges faced by aging projects in Maven Central due to missing dependencies. Our analysis, using the Maven Explorer[4] indexer, confirms that older packages are indeed more prone to dependency resolution failures, with the lack of Transport Layer Security (TLS) in repositories being a frequent cause. This trend poses a significant risk, especially for open source projects that form an integral part of the software ecosystem. We recommend that developers and maintainers of popular open source projects consider uploading new versions of their libraries to multiple repositories. Furthermore, this research underlines the importance of active maintenance and the potential risks associated with dependencies from repositories requiring authentication, such as the GitHub package registry. Our findings emphasize the need for better dependency management practices and highlight areas for future research. The insights gained from this study contribute to enhancing the longevity of software projects in the Maven ecosystem, with some extension to other package management systems.

Acknowledgments

This project would not have been possible without Dr.Ir. Sebastian Proksch. Thank you for all the advice you have given over the course of this research and for providing access to the large dataset of Maven artifacts.

Many thanks to my parents and friends who supported me while I was working on this project.

Special thanks to Mathijs van der Schoot, who tested my data collection program and in this way provided a large dataset which helped a lot for my experiment. I had the pleasure of working with Vivian Roest, Jelle Sandifort and Yifei Lu, who all gave invaluable feedback and insights during my research.

References

[1] S. Greene, “JCenter Shutdown Impact on Gradle Builds,” *Gradle*, Feb. 2021. [Online]. Available: https://blog.gradle.org/jcenter-shutdown

[2] P. Boldi, “How network analysis can improve the reliability of modern software ecosystems,” in *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, 2019, pp. 168–172.

[3] F. C. Y. Benureau and N. P. Rougier, “Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions,” *Frontiers in Neuroinformatics*, vol. 11, 2018. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fninf.2017.00069

[4] “maven-explorer,” Jan. 2024, [Accessed 23. Jan. 2024]. [Online]. Available: https://github.com/cops-lab/maven-explorer

[5] “Maven Repository: Repositories,” Jan. 2024, [Accessed 28. Jan. 2024]. [Online]. Available: https://mvnrepository.com/repos

[6] “NuGet Gallery | Home,” Jan. 2024, [Accessed 28. Jan. 2024]. [Online]. Available: https://www.nuget.org

[7] “PackageManagement - Debian Wiki,” Jan. 2024, [Accessed 28. Jan. 2024]. [Online]. Available: https://wiki.debian.org/PackageManagement

[8] S. Authors, “Requirements,” *Maven Central Repository Documentation*, Jan. 2024, [Accessed 22. Jan. 2024]. [Online]. Available: https://central.sonatype.org/publish/requirements/#sufficient-metadata

[9] “Apache Maven Dependency Plugin – dependency:resolve,” Oct. 2023, [Accessed 28. Jan. 2024]. [Online]. Available: https://maven.apache.org/plugins/maven-dependency-plugin/resolve-mojo.html

[10] “Apache Kafka,” Jan. 2024, [Accessed 24. Jan. 2024]. [Online]. Available: https://kafka.apache.org/intro

[11] “GitHub Search Results for “path:**/pom.xml jcenter.bintray.com”,” Jan. 2024, [Accessed 28. Jan. 2024]. [Online]. Available: https://github.com/search?q=path:**/pom.xml+jcenter.bintray.com&type=code

[12] E. al. Eric Redmond and K. H. Marbaise, “Maven – POM Reference,” Dec. 2019, [Accessed 24. Jan. 2024]. [Online]. Available: https://maven.apache.org/pom.html#Dependencies

[13] J. L. Harrington, *Relational Database Design and Implementation, 4th Edition*. Morgan Kaufmann, Apr. 2016. [Online]. Available: https://www.oreilly.com/library/view/relational-database-design/9780128499023

[14] J. van Zyl and E. Redmond, “Maven – Maven Documentation,” Aug. 2009, [Accessed 28. Jan. 2024]. [Online]. Available: https://maven.apache.org/guides

[15] “Sample size calculator - CheckMarket,” Jun. 2021, [Accessed 28. Jan. 2024]. [Online]. Available: https://www.checkmarket.com/sample-size-calculator/#sample-size-margin-of-error-calculator

[16] cops lab, “Franz,” Jan. 2024, [Accessed 24. Jan. 2024]. [Online]. Available: https://github.com/cops-lab/franz

[17] C. Inc., “parallel-consumer,” Jan. 2024, [Accessed 24. Jan. 2024]. [Online]. Available: https://github.com/confluentinc/parallel-consumer

[18] nkolev92, “NuGet Warning NU1803,” Jan. 2024, [Accessed 27. Jan. 2024]. [Online]. Available: https://learn.microsoft.com/en-us/nuget/reference/errors-and-warnings/nu1803

[19] D. K. Julian Andres Klode, Michael Vogt, “apt-get Manual Page,” Jan. 2024, [Accessed 27. Jan. 2024]. [Online]. Available: https://salsa.debian.org/apt-team/apt/-/blob/main/doc/apt-get.8.xml

[20] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, “A comprehensive study of bloated dependencies in the Maven ecosystem,” *Empir. Software Eng.*, vol. 26, no. 3, pp. 1–44, May 2021.

[21] C. Walls, *Spring in Action, Sixth Edition*. Riverside, NJ, USA: Simon and Schuster, 2022.

[22] M. Veytsman, “How to take over the computer of any Java (or Clojure or Scala) developer,” *Max.Computer Blog*, Oct. 2017, [Accessed 23. Jan. 2024]. [Online]. Available: https://max.computer/blog/how-to-take-over-the-computer-of-any-java-or-clojure-or-scala-developer

[23] Sebu, “How to disable maven blocking external HTTP repositories?” *Stack Overflow*, Jan. 2024, [Accessed 23. Jan. 2024]. [Online]. Available: http://web.archive.org/web/20240122233222/https://stackoverflow.com/a/67002852

[24] V. Roest, “An analysis of Java release practices on GitHub,” Jan. 2024, [Accessed 24. Jan. 2024, TU Delft Bachelor thesis].

A Relational Database Schema

```sql
CREATE TABLE artifact (
   id               BIGSERIAL PRIMARY KEY,
   artifact_identifier VARCHAR(255),
   group_identifier VARCHAR(255),
   packaging        VARCHAR(255),
   repository       VARCHAR(500)
);

CREATE TABLE artifact_release (
   id           BIGSERIAL PRIMARY KEY,
   release_date TIMESTAMP(6) WITH TIME ZONE,
   version_string VARCHAR(500),
   artifact_id  BIGINT REFERENCES artifact,
   m2directory_id BIGINT UNIQUE,
   CONSTRAINT uk_artifact_release UNIQUE (artifact_id, version_string)
);

CREATE TABLE explorer_result (
   id                           BIGSERIAL PRIMARY KEY,
   created_at                   TIMESTAMP(6) WITH TIME ZONE,
   explorer_stacktrace          VARCHAR(1000000),
   is_release_date_found_automatically BOOLEAN,
   kafka_lane                   SMALLINT
      CONSTRAINT explorer_result_kafka_lane_check CHECK ((kafka_lane ⩾ 0) AND (kafka_lane
          ⩽ 2)),
   kafka_topic                  SMALLINT
      CONSTRAINT explorer_result_kafka_topic_check CHECK ((kafka_topic ⩾ 0) AND
          (kafka_topic ⩽ 2)),
   processed_by_explorer_at     TIMESTAMP(6) WITH TIME ZONE,
   artifact_release_id          BIGINT REFERENCES artifact_release
);

CREATE TABLE error_summary (
   id              BIGSERIAL PRIMARY KEY,
   message         VARCHAR(500055),
   explorer_result_id BIGINT REFERENCES explorer_result
);

CREATE TABLE m2_directory (
   id               BIGSERIAL PRIMARY KEY,
   m2_path          VARCHAR(3060),
   artifact_release_id BIGINT UNIQUE REFERENCES artifact_release
);

ALTER TABLE artifact_release
   ADD CONSTRAINT fk_artifact_release_m2directory_id FOREIGN KEY (m2directory_id)
       REFERENCES m2_directory;

CREATE TABLE m2_file (
   id          BIGSERIAL PRIMARY KEY,
   content     VARCHAR(500055),
   filename    VARCHAR(255),
   m2directory_id BIGINT REFERENCES m2_directory
);

CREATE TABLE missing_dependency (
   id           BIGSERIAL PRIMARY KEY,
   error_summary_id BIGINT REFERENCES error_summary
);

CREATE TABLE error_path_element (
   id                BIGSERIAL PRIMARY KEY,
   index_in_path     SMALLINT,
   artifact_release_id BIGINT REFERENCES artifact_release,
   missing_dependency_id BIGINT REFERENCES missing_dependency
);
```

B Most commonly missing dependency elements

maven-downloader-plugin	org.apache.maven.plugins	40404
glide	com.github.bumptech.glide	962
annotation	androidx.annotation	943
appcompat	androidx.appcompat	938
hilt-android	com.google.dagger	807
wasp-model_2.11	it.agilelab	765
core-ktx	androidx.core	693
booster-android-gradle-compat	com.didiglobal.booster	638
network	me.proton.core	608
network-data	me.proton.core	600
wasp-core_2.11	it.agilelab	575
s2util	org.seasar.util	553
material	com.google.android.material	548
booster-android-gradle-api	com.didiglobal.booster	469
booster-api	com.didiglobal.booster	453
common-wrapper	com.bihe0832.android	448
common-about	com.bihe0832.android	394
framework	com.bihe0832.android	379
lib-wrapper	com.bihe0832.android	372
user-data	me.proton.core	364
repository	com.android.tools	357
common-feedback	com.bihe0832.android	357
yinhesdk_ui	com.github.vhall.sdk.yinhesdk	357
yinhesdk_baselib	com.github.vhall.sdk.yinhesdk	357
common-photos	com.bihe0832.android	350
constraintlayout	androidx.constraintlayout	327
common-image	com.bihe0832.android	315
common-list	com.bihe0832.android	315
presentation	me.proton.core	313
wasp-consumers-spark_2.11	it.agilelab	311
recyclerview	androidx.recyclerview	300
viewbinding	androidx.databinding	298
sdklib	com.android.tools	281
common-coroutines	com.bihe0832.android	280
common-webview	com.bihe0832.android	280
gifdecoder	com.github.bumptech.glide	274
micronaut-build	io.micronaut.build.internal	245
common-ace-editor	com.bihe0832.android	245
common-network	com.bihe0832.android	245
common-praise	com.bihe0832.android	245
kotlinx-coroutines-core-iosarm64	org.jetbrains.kotlinx	236
fragment	androidx.fragment	236
support-annotations	com.android.support	236
kotlinx-coroutines-core-iosx64	org.jetbrains.kotlinx	228
common-debug	com.bihe0832.android	228
vhall-module-core	com.github.vhall.android.base	221
tgbotapi.core-js	dev.inmo	220
gradle-toolkit	xyz.unifycraft.gradle	219
offline-acquirer-base	com.opayweb	217
common-data-center	com.bihe0832.android	214
coil-base	io.coil-kt	212
common-splash	com.bihe0832.android	210
challenge-data	me.proton.core	196
docker-controller-scala-core_2.13	com.github.j5ik2o	196
lifecycle-viewmodel-ktx	androidx.lifecycle	189
legacy-support-v4	androidx.legacy	188
binding	com.hi-dhl	188
exifinterface	androidx.exifinterface	185
lib-uiutils	com.bihe0832.android	184
docker-controller-scala-core_3	com.github.j5ik2o	182
coil	io.coil-kt	182
fragment-ktx	androidx.fragment	180
appcompat-v7	com.android.support	177
docker-controller-scala-core_2.12	com.github.j5ik2o	175

Table 2: Top 64 artifacts that are most often missing, causing the dependant to fail dependency resolution.

C Proportion of artifacts with missing dependencies per release year

Year	Proportion
2006	0.0624
2007	0.0866
2008	0.0692
2009	0.0342
2010	0.0245
2011	0.0285
2012	0.0237
2013	0.0236
2014	0.0393
2015	0.0346
2016	0.0253
2017	0.0170
2018	0.0133
2019	0.0074
2020	0.0060
2021	0.0047

Table 3: Proportion of artifacts with missing dependencies per release year to all found artifacts.