# An analysis of Java release practices on GitHub

**Vivian Roest**[1]

**Supervisor(s): 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 28, 2024

Name of the student: Vivian Roest
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Casper Poulsen

## Abstract

This paper examines the release practices of Java Maven Repositories on GitHub. Most prior research in this vein has been done on Maven Central, the largest Maven package repository. However, GitHub hosts 15.5 million Java repositories, and is left untapped. Additionally of interest is the fact that GitHub provides a competitor to Maven Central, GitHub packages. To this end, the paper establishes an index of all Java repositories on GitHub. Furthermore, this dataset also includes Maven configuration (`POM.xml`) files. Additionally, an in-depth analysis is done of a sample of 500 000 of those 15.5 million repositories. This sample ended up containing 170 798 Java Maven repositories that had those `POM.xml` files. In this sample we discovered that of those 170 798, 6 507 ($\approx 3.8\%$) had set up distribution configuration. Maven Central ended up being the most popular but GitHub packages and others ended up being quite popular as well. In the external repositories configured in those Java projects we notice a distinct lack of GitHub packages, other repositories were still present. We theorize that the lower popularity of GitHub packages is because it requires authentication, which is not trivial to set up. We discuss several approaches that can improve this situation.

## 1 Introduction

GitHub is the one of the most popular online code hosting platforms, it hosts a staggering 284 million public repositories [1]. This makes it an extremely attractive source of data to research what developers are doing in the wild. A lot of research into the practices of Java developers is customarily done using the data from *Maven Central* [2–7], the largest Java artifact repository[1]. Some studies [8, 9], do use GitHub rather than Maven Central. However, those still only analyse a maximum 34 560 repositories which is a fraction of the 15.5 million Java repositories available[2].

Therefore, there is this large untapped set of real world data on how developers use Java. Specifically what is interesting are the unique qualities that set GitHub apart from other sources. One such quality is that, unlike Maven Central which only hosts libraries, GitHub contains a lot of applications and personal projects. Additionally, GitHub potentially disrupts the centrality of Maven Central by not just being a platform to host Java source code, but also by hosting Java artefacts with GitHub packages[3]. GitHub packages is essentially a direct competitor to Maven Central.

This GitHub package repository also brings along with it some unique challenges. One of them is the fact that to download any package from the repository you need to configure credentials in your Maven config for the specific repository ID[4]. These GitHub repository URLs with their IDs are unique *per dependency* so if one has a lot of (transitive) dependencies on GitHub packages this can present quite the hassle to a developer.

Furthermore, GitHub also provides a Continuous Integration (CI) solution called GitHub Actions[5] which might provide us with a way into seeing how developers actually end up publishing their packages, and how specifically they deal with the authentication issue.

This leads us to the main research question (**RQ1**) of this paper: "What are the Maven release practices on GitHub?" Which can be split into the following subquestions:

**RQ2** Can we make a dataset of Java repositories on GitHub?

**RQ3** Are these projects released, and where are they released?

**RQ4** What is their use of external repositories?

**RQ5** How is authentication for releasing packages to distribution repositories realized?

The main contributions of this paper are a dataset of all Java repositories on GitHub, an analysis of where developers release their packages (e.g. Maven Central or GitHub), the usage of external (not Maven Central) repositories for downloading Java dependencies, and an investigation on the release practices of Java Maven projects.

For the dataset it is not only the data that is important but also providing a scalable, extendable method of obtaining this, as this could also be the starting point of research questions not considered in this paper.

First the methodology will be outlined, followed by the results. Afterwards a section on responsible research follows. Then the results and potential future work will be discussed to then end with a conclusion.

## 2 Methodology

This section will discuss the method and implementation of the pipeline that was created to answer the research questions. We will first go into what data we need and how we retrieved it in Subsection 2.1. Afterwards in Subsection 2.2 we will explain how and what we analysed exactly.

Both the implementation of the pipeline as the analyser are implemented in the Rust programming language because it provides well-performing code, ways to create fault-tolerant programs (specifically the error handling), and because of a familiarity of it with the author(s).

### 2.1 The Dataset

To properly answer the research questions we need to gather a certain set of data. This data needs to include a few things.

Firstly, for **RQ2** we simply want an index of all Java repositories on GitHub, similarly to how Maven Central provides an index of all its packages[6]. We also want to do this in a scalable efficient way, and ensure it can be used by others. Secondly, for research questions **RQ3** and **RQ4** we need to determine if a certain project releases its packages and if it uses

[1]https://mvnrepository.com/

[2]As determined by this paper

[3]https://github.com/features/packages

[4]See https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-apache-maven-registry

[5]https://github.com/features/actions

[6]https://maven.apache.org/repository/central-index.html

any external repositories. For this information we can utilize "POM.xml" files[7]. These POM files define dependency and project information for Maven Java Projects[8]. Finally, for **RQ5** we will fetch the GitHub workflow files of each repository that has releases, so we can inspect them to see how they realize their releases.

There do exist some prior attempts at mining GitHub as a whole such as GHTorrent[10], and GitHub Archive[9]. However, both are more focused on the stream of GitHub events rather than the data contained within the repositories. Therefore, these sources of GitHub data do not contain the 'POM.xml' or workflow files we need. Additionally, GHTorrent seems to sadly be unavailable since July 2019 according to its README[10]. Furthermore, some research on Python notebooks [11] done at Microsoft itself leverages the fact they have direct access to GitHub's raw backend. This is because they are the owners of GitHub. Unfortunately Microsoft does not share this data.

Interestingly, the Rust programming language, actually already has some precedence regarding this:

> We compile and test every single crate on crates.io and every single Rust repository on GitHub with a Cargo.lock file using the new version of Rust before releasing it.
> — Mara Bos[12, p. 4]

This program is called 'Crater'[13] and fetches similar metadata for Rust projects as the kind of data we set out to look for our Maven GitHub projects. We will turn to this project to see if we can make it fetch the data we need.

### 2.1.1 Indexing all Java projects on GitHub

GitHub contains a lot of repositories, around 284 Million[1]. So indexing these all is non-trivial, and requires a solution that is fault-tolerant and efficient. To start off with we turn to the aforementioned 'Crater' and examine their approach to help us make our own modified version fit for our purpose.

Crater's indexer works as follows: It uses GitHub's `/repositories` API endpoint[11] to request a list of (all) repositories. This list is paginated and chronological. Then for each repo on that list it does a GraphQL query to ascertain, if, firstly, the repository contains Rust code and secondly, the repository is not a fork. If both conditions are met it will check if a `Cargo.lock` (Rust's version of a `POM.xml`) file is present through the raw file API[12] and save that information in a CSV list. Which contains a hashed identifier, the repo path and a boolean indicating the presence of the `Cargo.lock`. Afterwards this data is used to clone all repositories and run a certain test suite on them to check if they compile, however, this part of Crater is not as relevant to us.

Initially some straightforward changes were made to this. Namely, changing it to search for Java instead of Rust, and looking for `POM.xml` instead of `Cargo.lock` files. Additionally, the code was also updated to use a newer version of Rust and some small refactors were made to make the codebase easier to work with.

However, as there are significantly more Java repositories than Rust ones a bigger rewrite ended up being necessary for our purpose. There are three additional major ways we differ from Crater's implementation (beside the general refactor and updating of the code). Firstly, our implementation searches recursively through *all* `POM.xml` files instead of only top-level ones, but more on that in the next paragraph. Secondly, we made error handling more robust. We did that because the indexer is supposed to run for days on end to gather all the data needed, and therefore it should not crash randomly and require manual intervention. The update was needed because while Crater could deal with certain errors it seemed the GitHub API had changed the way it reports those errors, this was of course important to change for our implementation. The other things considered were mainly: malformed responses from GitHub, unexpected status codes from GitHub, and spurious network failures. Thirdly, GitHub has a rate limit of 5000 API requests per hour[13]. Which causes problems when trying to index millions of repositories. For Crater this was not as much of a consideration as the amount of Rust repositories is comparatively smaller. Because the 5000 API request limit is *per API token* and not per IP address the program accepts an array of GitHub tokens. It cycles through these tokens whenever encountering an error code that indicates the program is being rate limited. Whenever it has wrapped around to the first token the program waits one minute to not overload the API unnecessarily.

### 2.1.2 Downloading all POM files

After having compiled a list of Java repositories on GitHub, to find answers to our research questions we need the `POM.xml` files from these repositories. Because these files are not required to be top-level and that a project may have multiple we need to search through all files and folders within a repository. To this end we use GitHub's `trees` endpoint[14], which gives as the Git tree at a specific commit of a repository. Because we do not need historical data we simply grab the latest version of the main branch, this can be done by specifying `HEAD` as our git ref.

Because a repository might have many `POM.xml` files these files are fetched asynchronously and in parallel, as to not create a bottleneck. This ensures we can fetch as many files as the rate limit allows us to.

### 2.1.3 GitHub Workflow Collection

As we are also interested in if and how repositories use GitHub workflows we use a similar approach as described in Subsection 2.1.2, to download GitHub workflow files if

[7]Project Object Model, see also: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

[8]These are similar to `build.gradle`, `package.json`, and `cargo.toml` files from Gradle, NPM and Cargo respectively.

[9]https://www.gharchive.org/

[10]https://github.com/ghtorrent/ghtorrent.org

[11]https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28

[12]Namely `raw.githubusercontent.com/`

[13]https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28

[14]https://docs.github.com/en/rest/git/trees?apiVersion=2022-11-28

present. Unfortunately, we have not analysed these workflow files programmatically given the limited time available for this research. However, this was also done to show that the scraper is relatively easy to extend. The complexity of analysis is higher but of course also depends on the data that is desired.

## 2.2 Analysis

Finally, to extract the statistics we want to know we need to parse the `POM.xml` files. However, it is not necessarily that straightforward, as POM files can inherit from other POM files, so POM files are not self-contained. To address this Maven has a command-line utility called 'effective pom', which resolves a POM file into an effective POM, which resolves parent POMs, interpolates variables and even can run plugins to fully full in all information needed to build a project. We run this utility for every POM file in a repository and then merge the results to get one 'report' per repository.

Unfortunately generating these 'effective POM' files takes quite a bit of time, partially because of the network requests it may make, but also simply the execution of the Maven commandline tool can take a while. To improve the throughput of analysis we use multithreading to spread out the work on all available CPU cores. The pipeline also has an option to *not* generate this effective POM (but still read older ones that were generated earlier), for testing or if one would not need the full effective POM for their analysis. This speeds up the analysis significantly.

The specific keys we extract from the final POM files are as follows:

- The top-level repositories in the POM file, which are the ones used for dependencies. To see if projects use external repositories for their dependencies.
- The repositories under `distributionManagement`, to see where and if a repository is released.

This report contains the following information

- Total number of repositories analysed
- Per distribution repository how many GitHub repositories use it
- Per external repository how many GitHub repositories use it
- The flat number of repositories that have external repositories configured
- A list of repositories that have distribution repositories configured
- A list of errors that occurred during processing

Additionally, we also calculate per hostname how many distinct repositories URLs there are, this is to ascertain the decentralized nature of the repositories in the dataset.

# 3 Results

This section shows the results collected from the pipeline described in Section 2. We will start with a look at the complete dataset that was gathered in Subsection 3.1, to answer **RQ2**.

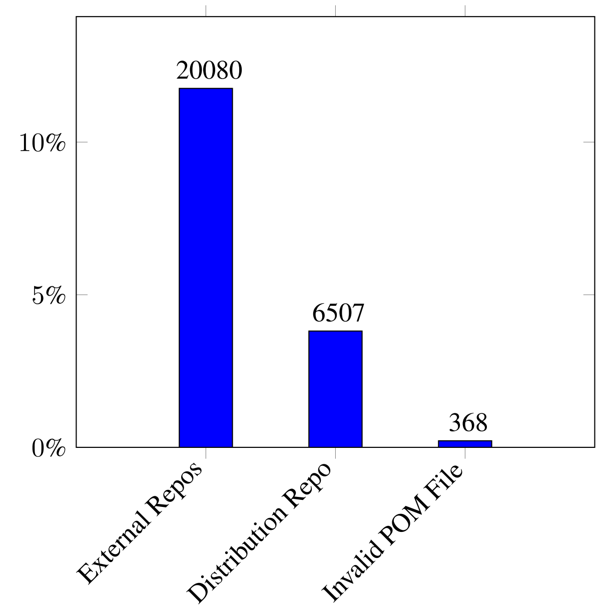

Figure 1: The percentage of repositories that have external repos, distribution repos or invalid POM files out of a total of 170 798. The y-axis shows percentage, the numbers on the bars are the absolute number of repositories in each category.

Afterwards, we will look at the popularity of both distribution and external repositories in Subsections 3.2 and 3.3, to answer **RQ3** and **RQ4** respectively. Finally, we will have a look at the gathered workflow files for **RQ5** in Subsection 3.4.

## 3.1 The Dataset

While creating the dataset of all Java repositories on GitHub we ended up with a staggering 15.5 million repositories. This statistic is already of note because GitHub's search feature claims that there are only 4.1 million Java repositories[15]. It could be that GitHub only reports repositories that have $> 50\%$ of Java files, but that's purely a conjecture. GitHub seemingly does not provide information on how this number is established.

We ended up using a random sample of 500 000 of those 15.5 million to analyse. However not all of these ended up containing `POM.xml` files, so we ended up with 170 798 repositories we actually analysed. This sample was entirely randomly selected with a fixed seed programmed into the program to ensure reproducibility given the same initial data. The reason for not analysing all repositories is mostly because of the runtime of the Maven commandline tool to make an effective POM, which could take up to three seconds per `POM.xml` file.

## 3.2 Distribution Repositories

To answer **RQ3** we will now look at which distribution repositories are most popular and how prevalent using them and thus releasing dependencies is.

[15]Data from GitHub search retrieved on 2023-11-24

| number of uses | url |
|---|---|
| 2362 | `https://oss.sonatype.org/service/local/staging/deploy/maven2` |
| 382 | `https://repository.apache.org/service/local/staging/deploy/maven2` |
| 193 | `https://s01.oss.sonatype.org/service/local/staging/deploy/maven2` |
| 103 | `https://repo.spring.io/libs-release-local` |
| 101 | `https://repository.jboss.org/nexus/service/local/staging/deploy/maven2/` |
| 93 | `https://repo.jenkins-ci.org/releases/` |
| 67 | `http://oss.sonatype.org/service/local/staging/deploy/maven2/` |
| 55 | `dav:https://google-maven-repository.googlecode.com/svn/repository/` |

Table 1: Most Popular Maven distribution repositories, exact URLs

| number of uses | distinct repositories | url |
|---|---|---|
| 2362 | 15 | `oss.sonatype.org` |
| 393 | 2 | `repository.apache.org` |
| 245 | 238 | `maven.pkg.github.com` |
| 205 | 5 | `s01.oss.sonatype.org` |
| 106 | 105 | `api.bintray.com` |
| 105 | 3 | `repo.spring.io` |
| 105 | 3 | `repo.jenkins-ci.org` |
| 103 | 2 | `repository.jboss.org` |

Table 2: Most Popular Maven distribution repositories, grouped by hostname

To start off with we can look at Figure 1 to see that the number of Java Repositories (that have POM files) from GitHub that use distribution repositories is 6 507 of the 170 798 which translates to $\approx 3.8\%$. This figure may seem on the one hand small, but as we expected most repositories on GitHub are not libraries but rather applications.

We also gathered which distribution repositories specifically were most popular. We split this into two tables. First by exact URL as we can see in Table 1. Secondly, also by hostname as seen in Table 2 to get a more generalized overview.

### 3.2.1 What do these URLs mean?

This section gives a summary of the most popular URLs we are seeing. We can see that the most popular repository URL is `oss.sonatype.org`, this makes a lot of sense as this is the URL used by Maven Central to publish packages. Additionally, we also see `s01.oss.sonatype.org` ranked highly which is the new URL to be used for Maven Central[16]. Both of these URLs for Maven Central have a low number of distinct URLs as there might be some variations with a leading backslash or not, http or https, and with using for example the snapshot repository URL like `oss.sonatype.org/content/repositories/snapshots` to get newer versions of dependencies. If we also look at the specific URLs we also see that some people still use the unsecure `http` version of the URL.

Next up, we see that `repository.apache.org` is another very popular repository. This repository contains the artefacts of various Apache projects, and as many Apache projects inherit from the Apache parent POM[17] they automatically include this distribution URL.

We also see that GitHub's own package repository is relatively popular sitting at third place, with 245 Java repositories using it. Here we also see the high number of distinct URLs as for every GitHub package you get a new Maven repository URL.

Similarly, for 'bintray' which was a competitor for Maven Central to distribute one's artefacts. It also uses unique URLs per dependency. It is less popular than GitHub, but that is also because of the fact it is discontinued[18].

Now we turn to `repo.spring.io` which is both used by the Spring[19] developers to publish their packages but was also used as an artefact cache until 2020[20] which allowed unauthenticated users to push their packages to it. As this has now discontinued it is also falling out of favour, but plenty of old repositories still have it configured.

Jenkins's is a Continuous Integration platform[21], the reason it is relatively high on the list of distribution repositories is that after requesting permission[22] a developer is allowed to push their plugins for Jenkins to their repository.

Finally, `repository.jboss.org` is a shared repository by all members of the JBoss community[23], which allows them to publish packages onto it. As again it is a relatively open repository it has quite a bit of use.

[16]https://central.sonatype.org/news/20210223_new-users-on-s01

[17]https://github.com/apache/maven-apache-parent/

[18]https://bintray.com

[19]https://spring.io

[20]https://spring.io/blog/2020/10/29/notice-of-permissions-changes-to-repo-spring-io-fall-and-winter-2020/

[21]https://jenkins.io

[22]https://www.jenkins.io/doc/developer/plugin-governance/managing-permissions/#release-permissions

[23]https://developer.jboss.org/docs/DOC-11377

## 3.3 External Repositories

To answer **RQ4** we will now look into the popularity of external repositories in our sample.

We also gathered which external repositories were most popular. Again split into two tables. The exact URLs in Table 3 and grouped by hostname in Table 4 which also again has the number of distinct repositories per hostname that are actually used, we can see again that some URLs are much more fragmented than others.

Again we will look at some of these repositories individually to understand what they host and why they rank so high.

### 3.3.1 What do these URLs mean?

First up we see `repo.spring.io` being the most used external repository. Some Spring dependencies are just available on Maven Central but apparently not all, it being one of the more popular Java frameworks it makes sense to see it so high in popularity. We see some number of distinct repository URLs for spring, this is because spring has various URLs for snapshots, milestones, releases all per categories like plugins, libs, etc.

Next up, `oss.sonatype.org` and by extension `repo1.maven.org` are the URLs for Maven Central, ordinarily you would not have to specify these URLs in your POM file as it is included manually but evidently some developers still do, this can again be for snapshots or other reasons. Specifically of note is that `repo1.maven.org` is the old URL for maven.

Moving on, `hub.spigotmc.org` is also very popular. This is a repository hosting plugins for the open source server software of the video game Minecraft[24]. When developing new plugins one might want to import other ones. As this video game is so popular, this ranks highly.

Furthermore, we see `maven.aliyun.com` which is the Maven repository for the Alibaba cloud provider[25], it presumably hosts or caches some packages for it, however the documentation appears to be all in Chinese[26], which is a language the author(s) of this paper do not speak, which made it hard to draw conclusions.

Additionally, we see that `jitpack.io` is also relatively popular. This is in essence a similar software solution to GitHub packages or bintray with one major difference: it references packages by Git URL and does not use Maven itself for publishing, hence the exclusion in Subsection 3.2. For example, one would reference a dependency that is hosted on jitpack as seen in Figure 2.

Finally, we see the now familiar bintray, apache and jboss repository. Whose popularity was already explained in the previous section on distribution repositories.

Finally, a notable exclusion this time around is GitHub's package repository with only 89 uses in our sample. This was not enough to show up in the list of most popular repositories. All of these 89 uses are for different packages, there is no singular popular dependency hosted on GitHub packages in our dataset.

[24]https://www.spigotmc.org/

[25]https://eu.alibabacloud.com/

[26]https://developer.aliyun.com/mirror/maven/

```
<repositories>
        <repository>
    <id>jitpack.io</id>
    <url>https://jitpack.io</url>
        </repository>
</repositories>
                        ⋮
<dependency>
  <groupId>com.github.User</groupId>
  <artifactId>Repo</artifactId>
  <version>Tag</version>
</dependency>
```

Figure 2: Usage example of jitpack.io, taken from: https://jitpack.io

## 3.4 GitHub Workflow Release Practices

If we now focus on just the GitHub Java repositories that release their packages on some distribution repository in Figure 3 we can see the proportion of those that use GitHub Workflows. As already mentioned in Section 2 no further analysis was done.

## 4 Responsible Research

This section describes how we worked towards producing this research responsibly. Firstly, this is achieved by designing for reproducibility and providing all data that was used to make replication easier. This can be read in Subsection 4.1 Secondly, the ethics of the research are considered in Subsection 4.2.

## 4.1 Reproducibility

An important step of the scientific method is the verification of results by retesting the hypothesis. This can only be done if the research is done in a reproducible manner.

To this end, the raw data that was scraped (the list of Java repos), as well as their POM files and the raw results are provided in a reusable machine-readable format of CSV and JSON files. Furthermore, a fixed seed was used to select the sample to make reproducing easier. Additionally, the implementation of the pipeline is also provided in a docker container to make it straightforward to run on other machines and eliminate the need for complex dependencies to run the pipeline. Where to find this data is described in Appendix A. The source code of the pipeline is also included there.

## 4.2 Ethics

There are a few ethical considerations with this research.

Firstly, as the scraper is indiscriminately downloading data from GitHub it might happen that we might (accidentally) download Personally Identifiable Information (PII) of a user that has upload that (possibly by mistake). However, as this data is already “in the wild” and we are not necessarily making it easier to find such data, the impact of this is limited.

Secondly, a malicious entity could take the software as created for this research to not just download what is essentially metadata but to “hunt” for private or secure information such

| number of uses | url |
|---|---|
| 4332 | `https://repo.spring.io/milestone` |
| 3040 | `https://repo.spring.io/snapshot` |
| 1521 | `https://oss.sonatype.org/content/repositories/snapshots` |
| 866 | `https://hub.spigotmc.org/nexus/content/repositories/snapshots/` |
| 773 | `https://dl.bintray.com/rabbitmq/maven-milestones` |
| 750 | `https://jitpack.io` |
| 695 | `https://oss.sonatype.org/content/groups/public/` |
| 638 | `https://repo.spring.io/libs-release` |

Table 3: Most Popular Maven external repositories, exact URLs

| number of uses | distinct repositories | url |
|---|---|---|
| 9488 | 54 | `repo.spring.io` |
| 3193 | 71 | `oss.sonatype.org` |
| 1250 | 4 | `maven.aliyun.com` |
| 1064 | 28 | `repo1.maven.org` |
| 1033 | 11 | `hub.spigotmc.org` |
| 1009 | 135 | `dl.bintray.com` |
| 908 | 29 | `repository.apache.org` |
| 866 | 5 | `repository.jboss.org` |
| 763 | 3 | `jitpack.io` |

Table 4: Most Popular Maven external repositories, grouped by hostname

as private keys that have accidentally been uploaded, or scraping data to train a Large Language Model (LLM) without consent of the users. But again, the scraper does not unearth any data that was not already public nor does it do anything to directly help such a use case.

In the end, this software, like lots of other software, could ultimately be used for unethical gains. But as the data we are scraping was publicly available to start with, and the data collected is mostly metadata the ethical concerns of this research are limited.

## 5 Discussion

In this section we will discuss the findings as presented in Section 3. First we will consider the dataset that we have created in Subsection 5.1. Secondly, we will go into the use of distribution and external repositories in Subsection 5.2. Thirdly, we will have a closer look at the GitHub packages repository in Subsection 5.3.

### 5.1 Dataset

For **RQ2** we created a dataset of all Maven Java repositories on GitHub. It ended up containing a list of 15.5 million repositories, with 3 960 369 repositories containing at least a top-level POM file. However, due to time constraints only 500 000 of the full 15.5 million were *recursively* scanned for POM files in subdirectories and GitHub workflow files. The tooling created as described in Section 2 does allow for incremental update of this data and will reuse data already on disk, if completely re-indexing. In any case this dataset is much more extensive than the ones presented in other papers[8, 9].

Even though we did not end up collecting and analysing all the data, it was retrieved relatively quickly, roughly within one week. Which shows it can be used in the future for other research even for other purposes wider than Java as the code was made to be extensible.

The most time-consuming part was the generation of the effective POM, this part is also more easily sped up as it's not limited by a rate-limit. For example, executing this on a high performance server could easily improve the analysis speed.

### 5.2 Distribution and External Repositories

The Maven Java dependencies' ecosystem is seemingly starting to move from a centralized ecosystem where every package is on Maven Central to a more decentralized one using a variety of external package repositories. As could be seen in Section 3. Especially the popularity for jitpack, bintray and GitHub packages are of note there. What also needs to be mentioned however is that quite a few of Maven repositories where one was allowed to upload arbitrary artefacts have been shutdown or limited. For example, we found that JFrog's bintray is one of the most popular external repositories but, it has been sunset[27]. For the time being read-only access remains, but this does showcase that depending on external repositories can incur significant risk if even one of the most popular ones gets deprecated. Even the spring repository allowed arbitrary uploads but shut this down as well a few years ago[28].

[27]https://jfrog.com/blog/into-the-sunset-bintray-jcenter-gocenter-and-chartcenter/

[28]https://spring.io/blog/2020/10/29/notice-of-permissions-changes-to-repo-spring-io-fall-and-winter-2020/

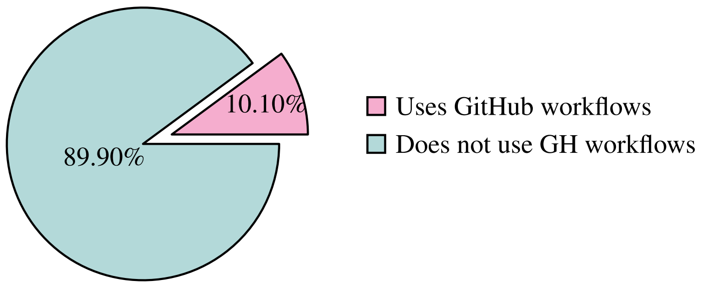

Figure 3: The number of repositories that use GitHub workflows compared to the number of repositories that are released (6 507).

In any case external repositories should not be discounted as does sometimes occur in research such as in [6]:

> "To ensure the consistency of our analysis, we discard the artifacts that depend on libraries hosted in external repositories" [6, p. 44]

Because they can remain a major source of dependencies for developers.

### 5.3 GitHub Packages Repository

In particular the GitHub Maven package repository is one of the more notable repositories as it is not aligned to a specific project like the Apache or Spring repositories for example, but a general purpose to be used by developers on GitHub. However, it seems quite popular to publish on but not popular yet to use. This is likely due to the fact that the GitHub package repository is unwieldy to use, given the need for authentication. Additionally, the fact that the GitHub URL and Maven ID need to be different for every dependency presents even greater barriers for developers to use it.

For developers, when choosing to release on GitHub packages, it is especially important to consider the downstream cost they are imposing on dependents of their packages. Because, now each user of that package needs to go through the trouble of configuring GitHub credentials. However, after searching through the dataset for the literal token '`maven.pkg.github.com`' we found that some developers however have discovered an intriguing workaround for this. As you can see in the snippets in Appendix B, some developers have, instead of relying on the authentication mechanisms presented by Maven, to a hardcoded public (hopefully scoped and read-only) token for GitHub Maven packages directly into the URL. This does immediately raise questions of security as intuitively hard-coding an access token seems to conflict with basic security principles. Although, as long as this token is properly scoped and only allows *read only* access to the very specific Maven GitHub package repository, the attack surface seems slim. It is good to mention despite it being relatively secure when properly configured, misconfiguration of these tokens would be dangerous. The actual usage of this technique seems very small, in our sampled dataset as described in Section 3, we only found two such cases.

Furthermore, researchers and developers can consider setting up a local proxy for GitHub packages to add a given authentication token to any request outbound to GitHub by simply adding a header or rewriting the URL. Especially for researchers needing to download large amount of packages this seems to be a useful solution.

Finally, both the Maven tooling and GitHub could improve this situation. If GitHub allowed developers to download packages from their repo without credentials (like how Maven Central does), this could alleviate some struggles. This would only be a band-aid however as other package repositories could come forth and encounter similar issues.

Alternatively, if Maven allowed developers to specify credentials for package repositories based on hostname instead of the arbitrary user-chosen IDs, the developer experience could be improved. Maven's tooling improving would be the preferred scenario as the 'ID' based system is very fragile in any case, as there is no convention in choosing them.

## 6 Future Work

In this section we will sketch some topics that could be considered for future work.

The dataset that was created and explored in this paper can also be used as a starting point for other research. For example, to delve deeper into the release practices of developers an analysis of how versioning is generally done could be of interest. Furthermore, seeing how much overlap there is with the GitHub dataset and Maven Central, and looking specifically at the practices of developers that host their source on GitHub to publish to Maven.

Another topic could be finding ways to improve the Maven tooling to suit a more decentralized nature. Can the Maven external repository IDs be changed into something more coherent or deprecated altogether, how much of a reliance does Maven itself have on these IDs. Likely inspiration and comparison to other package managers would also be prudent here, both Java focused ones like Gradle but also Docker. Docker specifically is notable in this regard as it mirrors the ecosystem of Maven in a way by having one registry that is most popular: Docker Hub (cf. Maven Central), but others like GitHub again as well.

As a more short term solution the effectiveness of a proxy that adds authentication to GitHub packages requests as outlined in Section 5, would also be an interesting avenue to evaluate. Because the authentication problem is not limited to the Maven part of GitHub packages seeing if other package managers also run into this problem and if such a proxy could also help there, might yield a broader perspective.

The research as presented in this paper only looks at the latest version of repositories. An additional angle that could be considered is a more historical look to see if developers use more or less external repositories over time. This research could consider both a global, for all repositories sense but also see if individual repositories switch away from external repositories due to fragility.

An important consideration for this however is that determining the "date" of a repository is not necessarily straightforward, does one use the initial creation date or the latest commit date, or even include multiple points in history for every repository.

## 7 Conclusion

In conclusion, this study has delved into the large number of Maven Java repositories on GitHub, shedding light on Maven release practices and the unique aspects of GitHub's role in the development landscape. This was done to answer the overarching research question of "What are the Maven release practices on GitHub?" For this we ended up looking into distribution repositories, which are online repositories where developers release their packages and external repositories which are repositories to fetch Maven dependencies from that are not Maven Central.

To gather the data required for the analysis a robust and extendable pipeline was created to fetch a list of all Java repositories on GitHub and their `POM.xml` files. We based this pipeline on Rust's Crater [13]. Furthermore, for selected repositories it also fetched GitHub workflow files if present. Additionally, to actually get a proper look at these `POM.xml` files the pipeline also created 'effective POM' files, which contain all data which might be inherited from parent `POM.xml` files and the like.

We ended up with a total list containing 15.5 million repositories. We then picked a random sample (with fixed seed) of 500 000 to analyse in depth. The sampled repositories did not all have `POM.xml` files so our final set of analysed repositories was 170 798 large.

For distribution repositories (Subsection 3.2), we identified 6 507 Java repositories (approximately 3.8% of the analysed subset. The most popular distribution repository ended up, unsurprisingly, being Maven Central. Though what was perhaps more surprising was that the third most popular distribution repository ended up being GitHub packages, with 245 repositories using it. Furthermore, the now deprecated 'bintray' also ranked highly, as well as the Spring and Jenkins repositories.

For external repositories (Subsection 3.3), we saw Spring being really popular, which makes sense given its widespread use. Furthermore, various snapshot URLs were also quite popular, as were specific projects like the Minecraft Server 'SpigotMC'. A notable omission was GitHub packages only being used 89 times for also 89 different dependencies. This indicates that actually consuming GitHub packages is as of now still unpopular. A repository that didn't show up under distribution but did under external was 'jitpack' which does not use the Maven tooling to publish packages and is therefore also an interesting case.

Now turning to GitHub workflows, we did see they were somewhat used, at around 10.10% of all the distribution repositories. But clearly not ubiquitously, further analysis of those might also be needed in the future as this research only touched the surface in that regard.

Looking closer at the numbers we obtained we saw that Maven Central is not as central as it might have used to be. With quite a few uses of different package repositories, both for publishing and consuming. Specifically GitHub packages was an interesting case, being so popular for publishing but not for using. The main reason is speculated to be the issue of authentication. We discussed various workarounds such as a proxy or even inlining authentication tokens.

## References

[1] GitHub. *Octoverse: The state of open source and rise of AI in 2023*. 2023. URL: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/ (visited on 11/24/2023).

[2] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository". In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 2014, pp. 215–224. DOI: 10.1109/SCAM.2014.30.

[3] Lina Ochoa et al. "Breaking bad? Semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study". In: *Empirical Software Engineering* 27.3 (2022), p. 61.

[4] Raula Gaikovina Kula et al. "Trusting a library: A study of the latency to adopt the latest maven release". In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. 2015, pp. 520–524.

[5] Amine Benelallam et al. "The maven dependency graph: a temporal graph-based representation of maven central". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 344–348.

[6] César Soto-Valero et al. "A comprehensive study of bloated dependencies in the maven ecosystem". In: *Empirical Software Engineering* 26.3 (2021), p. 45.

[7] Dimitris Mitropoulos et al. "The bug catalog of the maven ecosystem". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 372–375.

[8] Thomas Durieux, César Soto-Valero, and Benoit Baudry. "Duets: A Dataset of Reproducible Pairs of Java Library-Clients". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 2021, pp. 545–549. DOI: 10.1109/MSR52588.2021.00071.

[9] Phuong T Nguyen et al. “Focus: A recommender system for mining api function calls and usage patterns”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1050–1060.

[10] Georgios Gousios and Diomidis Spinellis. “GHTorrent: GitHub’s data from a firehose”. In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE. 2012, pp. 12–21.

[11] Fotis Psallidas et al. “Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML. NET Pipelines”. In: *ACM SIGMOD Record* 51.2 (2022), pp. 30–37.

[12] Mara Bos. *Do we need a “Rust Standard”?* 2022. URL: https://blog.m-ou.se/rust-standard/.

[13] *Crater: a tool to run experiments across parts of the Rust ecosystem*. URL: https://github.com/rust-lang/crater.

[14] Vivian Roest. *Data underlying the BSc project: ”An analysis of Java release practices on GitHub”*. 2024. DOI: 10.4121/67A790FE-B65A-4C30-AAE0-C5B2DC7E5D4D.V1.

## A Reproducibility

The source code and data collected can be found on the 4TU Research Data repository [14]. Additionally, the source code is also available on GitHub: https://github.com/NULLx76/maven_github_scraper.

## B Hardcoded GitHub Tokens

```
...
<repositories>
        <repository>
                <id>Central</id>
                <url>https://repo1.maven.org/maven2</url>
        </repository>
        <repository>
                <id>github-public</id>
                <url>https://public:&#103;
                    hp_Y6nRFazi9yNo0IMpxwTFIagW352c1539nyfn@maven.pkg.github.com/
                    kvalitetsit/*</url>
        </repository>
        <repository>
                <id>github-public1</id>
                <url>https://public:&#103;
                    hp_Y6nRFazi9yNo0IMpxwTFIagW352c1539nyfn@maven.pkg.github.com/
                    kvalitetsit/*</url>
        </repository>
        <repository>
                <id>github-public2</id>
                <url>https://public:&#103;
                    hp_Y6nRFazi9yNo0IMpxwTFIagW352c1539nyfn@maven.pkg.github.com/
                    kvalitetsit/*</url>
        </repository>
</repositories>
...
```

Figure 4: An example from https://github.com/KvalitetsIT/medcom-video-api/blob/master/pom.xml.

```
...
<repositories>
    <repository>
        <id>github-public</id>
        <url>https://public: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@maven.pkg.
            github.com/lblod/*</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
</repositories>
...
```

Figure 5: Another example from https://github.com/lblod/jsonld-delta-service/blob/master/pom.xml.