# EMSE-D-20-00026: Revision

Dear Editorial Office of *Empirical Software Engineering*,

We would like to submit a revision for our manuscript entitled **"A comprehensive Study of Bloated Dependencies in the Maven Ecosystem"**. We thank the reviewers for their detailed recommendations. We have addressed all their comments, which has improved the quality of our manuscript.

In this revision:

- We assess the limitations of static analysis to address the comments 1.5 and 1.6 of Reviewer #1. We add new results based on a new experiment where we run the test suite of the projects to evaluate the impact of DepClean when detecting bloated dependencies.
- We clarify the methodological aspects about our experimental setup for analyzing bloated dependencies, in particular when referring to the role of transitive dependencies.
- We add and discuss the missing references pointed out by Reviewer #1.
- We clarify the novel concepts in this manuscript with respect to our previous work related to the Maven Dependency Graph (MDG).
- We updated the qualitative results to reflect the pull requests that were answered after the submission of this paper, as suggested by Reviewer #2.
- We specify the number and types of the bloated dependencies in the Abstract and Conclusion sections, to accurately present our results.
- We remove the part in the discussion that refers to the motivations of dependency analysis for security, as suggested by Reviewer #3.

We have corrected all the minor remarks and typos that the reviewers brought to our attention. In the following pages, we give detailed answers to each of the reviewers' comments. The original texts from the reviewers are included in boxes, our answers follow the boxes. All changes are highlighted in blue in the resubmitted version of the manuscript (except typos).

In case of requiring any further information, please do not hesitate to contact us.

Sincerely yours,

César Soto-Valero

On behalf of Nicolas Harrand, Martin Monperrus, and Benoit Baudry

## Reviewer #1

> **Comment 1.1** "How about not naming the dependencies "bloated" (this has a connotation) and instead just for what they are, unused? This also follows nomenclature of earlier work."

We thank the reviewer for this suggestion. However, we believe that the term "bloated" applies in the context of this work. The content of our manuscript is in line with the state-of-the-art on software debloating, which the reviewer can consult in our updated living review on this topic: https://www.cesarsotovalero.net/software-debloating-papers. Notice that the term "bloated dependency" applies to both the unused dependencies that developers are aware of having included in the dependency tree, and those that are added to the classpath based on the Maven's dependency management mechanism. We prefer to keep the term "bloated" to refer to all types of unused dependencies.

> **Comment 1.2** "The abstract and conclusion draws too broad a conclusion based on the number of transitive dependencies being unused. However, as the paper also elaborates, developers largely do not have a direct influence on these. To draw an accurate instead of a misguiding, sensationalist finding, at the very minimum, the number advertised in the abstract and conclusion should differentiate between direct bloated and indirect bloated dependencies."

We have addressed this comment by specifying the percentage of each type of bloat (bloated-direct, bloated-inherited, and bloated-transitive) in the second paragraph of the Abstract and the second paragraph of the Conclusion of the manuscript. Besides, we have clarified the distinction between the number of direct and transitive dependencies removed, in the first sentences of the Summary of RQ3 and RQ4 (pages 32 and 36), which provides a more accurate presentation of our results.

> **Comment 1.3** "The article is missing two essential pieces of related work: [5] pre-empts the idea of unused dependencies (albeit in the Rust ecosystem); [4] is the foundational work on the security analysis mentioned in RW."

We thank the reviewer for this remark. We have cited [5] in the Introduction section, and we have referred to [4] and [5] in Section 7.1 (Related Work).

> **Comment 1.4** "This also reduces the claim of novelty somewhat, although to the best of my knowledge, doing this study in the Java ecosystem and validating with developers is indeed novel. (If one feels the need to stress this.)"

We found the work of Hejderup et al. [4] on vulnerable dependencies and its follow up paper [5] relevant for our work. PRÄZI constructs a fine-grained call-graph dependency network for the Rust ecosystem of libraries CRATES.IO. In [5], two applicable case studies on security vulnerability propagation and deprecation impact analysis are presented. DepClean uses an approach in the same spirit as the one implemented in PRÄZI. However, DepClean focuses on one specific problem: bloated dependencies, and we evaluate the soundness of the analysis performed with DepClean directly with developers through a qualitative study.

As far as we know, our study is the first that explores and consolidates the concept of bloated dependencies in the Maven ecosystem. This work is novel because it includes: 1) a large scale quantitative study of bloated dependencies, 2) a qualitative analysis of the feedback from developers

when suggesting the removal of such bloated dependencies, and 3) a tool that facilitates the detection and removal of bloated dependencies in Maven projects. Consequently, we have kept our novelty claims as we believe that a work similar to this in the Maven ecosystem has not been published before. We have added a brief discussion about the relevance of our contributions with respect to PRÄZI and other related works in the last paragraph of Section 7.1 (page 39).

> **Comment 1.5** "The criticism we received from reviewers is the fact that the Präzi analysis is not fully sound. DepClean is not sound, either, as it is missing dynamic function calls. However, there are excellent call graph generators that deal with this problem in Java (e.g., Wala). Why not use one of these? For me, this is not a reason to not accept this article. However, 1) the threat needs to be quantified extensively (at the minimum, have a look at Präzi for an inspiration on how we tackled this, what about invoke dynamic, reflection) or 2) a different call graph generator has to be used."

We thank the reviewer for raising this technical limitation, which we have discussed thoroughly in our manuscript (see last paragraph of Section 3.3.1, and Internal Validity in Section 6.2). We build DepClean on top of the Maven dependency-analyzer [9], which is actively maintained by the Maven team and officially supported by the Apache Software Foundation. The Maven dependency-analyzer addresses various limitations and pitfalls of static analysis of Java bytecode. It internally uses state-of-the-art bytecode analysis techniques that take into account some reflection mechanisms. For example, it captures all the dynamic invocations created from class literals by parsing the bytecodes in the constant pool of the classes. It also implements a customized bytecode parser that allows the identification of reflection calls from particular Java constructs, from a project towards its dependencies. However, it may still trigger false positives, which are due to native code, the usage of Java reflection APIs, or the usage of the Unsafe API. To mitigate this threat, we have added new configurable options to DepClean in order to ignore manually added dependencies or Maven scopes from the analysis.

As suggested by the reviewer, we have added more details in the last paragraph of Section 3.3, to explain the most important technical nuances of how DepClean handles dependency analysis in the case of reflection, invoke dynamic, and other Java dynamic mechanisms. We have also improved the description of its limitations in Section 6.2 (Threats to Validity).

> **Comment 1.6** "What about a strategy to flag a dependency as unnecessary when compiling and executing tests works?"

We thank the reviewer for this suggestion. When we prepared the pull-requests for the 30 projects in our qualitative study, we checked as part of our protocol that we could build the projects and successfully run the test suite, before running DepClean, as well as after generating a debloated build file. We have made the explanation of this procedure more explicit in the third paragraph of Section 4.2.2.

As a complementary validation of DepClean for this revision, we have performed new experiments on 10 popular projects. For these experiments, we executed the test suite of the additional projects with the version of the POM generated by DepClean, without bloated dependencies. The additional results obtained are presented in Table 7 and discussed in Section 6.2. We have included the data of these results as part of the complementary repository of this manuscript, which is available at `https://github.com/castor-software/depclean-experiments/tree/master/ts-experiments`.

We want to mention that, to validate DepClean using the testing strategy suggested by the reviewer, we need to use Maven projects that satisfy various conditions: 1) we can build them successfully, 2) they contain at least one bloated dependency, 3) they have a test suite that exercises the API members of their dependencies. Also, it has to be noted that, even if the project compiles and the tests pass, we cannot claim that the removed dependencies are not used with 100%

accuracy due to the limitations of bytecode analysis discussed previously and also due to other limitations such as the presence of flaky tests. Our qualitative study and the feedback provided by the developers of 30 Java projects aim to perform this type of validation.

> **Comment 1.7** "The article is very long. The concepts introduced in it are easy to grasp. I recommend shortening, but also understand if the authors do not want to follow-up on this."

We have shortened Section 2, keeping what we consider is essential for the readers that are not familiar with Maven. We have also made an effort to reduce the size of Section 3, while consolidating the key concepts that form the foundation of our work.

> **Comment 1.8** Introduction "To our knowledge there is no ..." Reference to Präzi? [5].

We added a reference to the work of Hejderup et. al. [5] in Section 1 (Introduction).

> **Comment 1.9** Introduction "mostly due to transitive dependencies" is unclear at this point.

We have clarified this sentence. We have edited the corresponding paragraph to make all the sentences in the past tense.

> **Comment 1.10** p22 l48 This is a very nice message, however it needs to be more nuanced. The default maven strategy is only suboptimal wrt. ensuring minimal dep inclusion. It might be optimal wrt other aspects. This aspect is never picked up in the discussion.

We thank the reviewer for this remark. The Maven dependency resolution strategy, which always picks the dependency that is nearest to the root of the tree [8], is a reasonable dependency selection criterion. This approach has several benefits, for example, it makes it easier for developers to control the dependency version used by declaring explicitly such dependency in the POM. It also provides a more natural dependency version update, by prioritizing the update of dependencies that are more likely to be used by the artifact (i.e., the root of the tree). We have clarified this point in Section 5.1 (Results) and referred to it in Section 6.1 (Discussion).

> **Comment 1.11** p27 very Maven-specific. How would it generalize?

Section 5.2.2 compares single and multi-module artifacts with respect to their number of bloated dependencies. We agree with the reviewer that the focus of this section is very Maven specific. We found this relevant since very few papers study multi-module projects, although it is a widely used architecture: 4,967 (51.5%) of artifacts in our dataset belong to multi-module projects.

This section is the result of our empirical observation when investigating the causes of dependency bloat. We want to mention that nobody has ever reported that multi-module is one of the main causes of bloat. Also, we want to notice that the construction of multi-module projects is not specific of Maven. For example, Gradle projects can be also architecture in a modular manner. We have clarified this in Section 5.2.2. We have also generalized the need of tools and user interfaces to help developer manage their inherited dependencies.

## Reviewer #2

**Comment 2.1** "In the threats to validity section the authors mention as part of External Validity to carry out a similar study for other package managers. However, I am wondering to which extent this is easy to do. Java libraries in Maven are based on a statistically typed programming language, while most other package managers are for dynamically typed languages (e.g. RubyGems for Ruby, npm for JavaScript, Cargo for Rust, ...) The dynamic nature of these languages will make the analysis much more challenging. I am also wondering to which extent the package metadata that contains dependency information is different across ecosystems, and to which extent this will make it more challenging to extend the analysis to other ecosystems. For npm packages in particular, the metadata is represented in JSON files, with an explicit distinction between runtime dependencies, development dependencies and optional dependencies. It is only the runtime dependencies that will actually be part of the installation/deployment. Hence, because of this separation I suspect that the phenomenon of bloated dependencies will be less prominent for npm and related package managers. This is just an intuition, though."

We agree with the intuition of the reviewer and share his/her thoughts about the challenges of replicating our results on other ecosystems.

**Comment 2.2** "The authors discuss at several points in the paper about Mavens dependency resolution mechanism (which is based on the nearest-wins strategy), and discuss why this mechanism is perhaps not the most optimal. It would have been nice if the authors could have provided concrete recommendations or suggestions about what would be a better resolution mechanism (in terms of benefits for reducing the amount of bloated dependencies which has been observed as being very high). For example, on Page 28L4-6 the authors mention "This calls for better tooling and user interfaces" but they do not provide any insights on what such tooling should look like."

We thank the reviewer for this remark. Today there is no way to test a build file, and POMs are very verbose. In this manuscript, we call for tools to manage build files, which is the place where developers handle their dependencies. The sentence: *"this calls for better tools and user interfaces..."* refers to improving the Maven dependency resolution mechanism by integrating the analysis of bloated dependencies as part of its resolution strategy, which will limit the bloat, the conflicts, and therefore facilitate the dependency management. We have clarified this idea in the last paragraph of Section 5.2.2. The analysis of concrete proposals to improve the resolution strategy of Maven is out of the scope of this work. Our suggestion is that the dependency resolution process should be highly customizable to meet the most specific user requirements. We do not aim at rejecting the default dependency resolution mechanism of Maven but rather in using the observation of bloated dependencies to make the reader notice that this mechanism is not optimal. In particular, our focus is on the minimization of the number of dependencies included in the classpath of the projects. We have clarified this point in Section 5.1 (Results) and referred to it in Section 6.1 (Discussion).

Beyond the points above, which address the reviewer's comment, we want to refer to the nuances of the Maven dependency management strategy. Maven resolves version conflicts by picking the dependency that is nearest to the root of the dependency tree [8]. This strategy is ordering dependent, and keeping order in a very large dependency graph can be a challenge. For example, it could happen that the dependencies of a newly added dependency change the order of the dependency tree, altering the versions used and causing the so called *dependency hell*.

Furthermore, there exist several ways to handle dependency resolution, ranging from manually selecting a specific version, or failing the resolution if there are several versions of the same dependency. For example, by default, Gradle resolves dependency conflicts by prioritizing the latest version of the dependency in the tree [6]. However, this strategy also has its own limitations,

since always favoring the latest versions could make projects more vulnerable to unknown bugs or zero-day attacks. As another example, the Apache Ivy™ [10] dependency manager is more configurable, but this flexibility comes with the price of making dependency management hard to reason about. These topics are out of the scope of this manuscript.

> **Comment 2.3** "I was wondering whether there are any other tools out there that are similar to DepClean (either for Maven or for other package managers) to detect and resolve bloated dependencies. I have not encountered the term "bloated dependencies" before, but I know that there are many tools available for dependency analysis in package managers (the tools tend to be different for different package managers), so I would assume that there would be at least some existing tool support for detecting unused dependencies. It would be nice if the authors could report on this. If no such tools exist, it would make the message of the paper even stronger."

There is no tool that is similar to DepClean, i.e., that analyzes the usage of direct and transitive dependencies in Java artifacts. We are not aware of similar tools for other package managers, with the exception of PRÄZI [5] for Rust. We have added a reference to PRÄZI in Section 1 (Introduction). In the Java ecosystem, the most related tool is the Maven dependency-analyzer plugin [9]. However, it analyzes dependency usage only at one level, i.e., the direct dependencies. We build DepClean with two main objectives: 1) to reason about dependency usages at the transitive level and 2) to generate a version of the POM that can be used directly to remove or exclude bloated dependencies.

> **Comment 2.4** "Is there a particular order in which these projects are listed? There does not seem to be. I suggest to list them in a specific order, whathever the authors consider most appropriate. E.g. ordered in decreasing number of commits, or stars, grouped by category, alphabetically ordered by project, ..."

The projects are listed in decreasing order according to their number of stars on GitHub. We have mentioned this in Section 4.2.2.

> **Comment 2.5** "Fig 12P24: Why does one observe star like patterns in the blue dots in this figure? How can this phenomenon be explained? What does the color of the blue dots mean? Some are darker and some are lighter ..."

The dot plot in Figure 12 shows the relation (in percentages) between the number of transitive dependencies and the number of bloated dependencies. Each point represents an artifact in our dataset. The number of dependencies in each artifact is a natural number $n \in \mathbb{N}$. Therefore, when computing the ratio of transitive dependencies that are bloated, the ratio of natural numbers creates the visual effect of having a star-shaped figure.

We decreased the opacity of the blue dots in order to add a sense of density to the figure in the presence of overlaps. We have mentioned this in Section 5.2.1 to facilitate the comprehension of the figure.

> **Comment 2.6** "Fig 13P25: I would suggest to aggregate all cases for depth 9 together with those for depth 10 or higher"

We have modified Figure 13 according to the suggestion of the reviewer.

**Comment 2.7** "P32L16: For the * pending PRs, it seems that 1 or 2 more have been accepted since the submission of this article."

The observation of the reviewer is correct. Thanks for noticing it. We have updated the results to reflect the three pull requests that have been accepted posterior to the submission of the manuscript.

**Comment 2.8** "P36L10-24: I am not sure if this paragraph about security management is really relevant or important. If yes, then why wasn't it reported by any of the discussions where the maintainers of those projects where pull requests were proposed ? Concerning the Equifax incident, would the absence of bloated dependencies have been able to avoid this problem? I do not think so..."

We have removed from the manuscript the paragraph that discusses the motivations of performing dependency analysis for security. But we still want to mention to the reviewer that security is a major concern for dependency management, as confirmed with our industrial partners.

**Comment 2.9** "References: Make sure the paper titles have the correct capitalisations, currently this is not the case for many references."

We have addressed this comment by carefully reviewing and capitalizing the titles of all the included references in the manuscript.

## Reviewer #3

### Major comments

**Comment 3.1** Background: "The authors may restrict this section to explain the important concepts in a paragraph."

We have reduced the size of Section 2 (Background). However, we did not restrict this section to a single paragraph because we do not assume that all the readers of *Empirical Software Engineering* are familiar with build automation tools in general and Maven in particular. Furthermore, we consider that Maven has some specific mechanisms that are essential to understand the content of the manuscript, as well as some unique terminology that may not be necessarily known by all readers. Therefore, in an effort to make this manuscript self-contained, we keep the definitions and examples necessary to understand the concepts of dependency management and build automation with Maven.

**Comment 3.2** Bloated Dependencies: "In 3.1 the authors describe "Novel Concepts". However, the authors already published a paper about the Maven Dependency Graph (MDG) [1] which is neither mentioned nor cited here. I'm a little confused because of that. Is this now a new novel concept of the MDG? Does this definition vary from the original one? The authors need to clarify this here., ..."

Thanks to the reviewer for noticing the missing references to our previous work on the MDG. We have added a sentence at the beginning of Section 3 (Bloated Dependencies) that cites our previous work and states its relation with this manuscript:

*"The analysis of bloated dependencies relies on our previous work that introduces the Maven Dependency Graph (MDG) [1]."*

We have also refactored the introduction of Section 3 to make a clear distinction between the MDG (previous work), and the novel concepts introduced in this manuscript.

> **Comment 3.3** Bloated Dependencies: "Figure 2 presents an example of the six types of dependencies that the authors describe. However, compared to the initial presentation in Figure 1, many edges changed. Especially the tree structure got lost and now all the edges start from the root jxls-poi. I do not fully understand how and why this transformation was performed. Please, give more detail and explain the necessary arguments"

Figure 1 in the manuscript presents an example of the Maven Dependency Tree (MDT) corresponding to the Jxls Maven library. In a MDT, there is a single type of edge, which represents the dependency relationship between each artifact (nodes) and its dependencies in the tree. With this representation, there is no explicit distinction between direct, transitive, or inherited dependencies, because the edges are not focused on the root artifact. There is also no distinction between used of bloated dependencies.

On the other hand, Figure 2 in the manuscript represents a Dependency Usage Tree (DUT), a novel concept that we described and formalized in Definition 4. We introduced this particular DUT data structure to analyze the relationship between a single artifact (the root) and every artifact that is present in its MDT. In a DUT, there are six types of edges, representing the type of dependency relationship (direct, inherited, or transitive) and the usage status (used or bloated) of the root node with respect to the rest of the nodes. Consequently, there are only arrows that go from the artifact towards its dependencies (nodes) in the DUT.

For the purpose of our manuscript, this combination of usage and type of dependency is a necessary information that allow us to reason about the different types of bloated dependencies. According to the suggestion of the reviewer, we have added additional details when explaining the construction of the DUT in Section 3.2 of the manuscript.

> **Comment 3.4** Bloated Dependencies: "Furthermore, the authors claim that a transitive dependency which they classified as bt (bloated transitive dependency) can be removed. I'd argue against that. Only because there is no direct call from the project to the transitive dependency, it does not necessarily mean that the dependency is not used. It might be used transitively, of course. This is even more critical as removing it may even cause runtime errors.

We consider a transitive dependency as bloated only if it is not used, whether explicitly via a direct call from the artifact, or indirectly through a transitive call from its dependencies. We determine the usage using static bytecode analysis, which is detailed in Algorithm 2. To do so, DepClean constructs a DUT that determines all the dependencies used, directly or indirectly by a Maven artifact.

For example, in Figure 1, when we determine if `commons-logging` is used by `jxls-poi`, we look at all the possible direct calls to API member of `commons-logging` by `jxls-poi`, as well as all the usages of `commons-logging` by the parts of `commons-jexl` that are used by `jxls-poi`, and all the usages of `commons-logging` by the parts of `commmons-jexl3` and the parts of `commons-beanutils` that are used by `jxls-poi`. This way, we consider all the edges in the DUT from `jxls-poi` to `commons-logging`, labelling the edges as bloated-transitive only if none of these artifacts use `commons-logging` to provide to `jxls-poi`.

In other words, our analysis of dependencies considers usages along the path of the MDT, and we characterize the usages on the DUT because the MDT does not have a concept that accumulates usages over a complete path in the tree. With this type of analysis, we reduce to a minimum the

chances of causing runtime errors due to the removal of a necessary transitive dependency. We have added a more detailed explanation about our analysis in Section 3.2 (Example).

> **Comment 3.5** Bloated Dependencies: Moreover, if it would be directly called from the project, I'd even advocate to explicitly declare this dependency in the root project.

We agree with the reviewer on this regard: explicitly declaring all the used dependencies is a good practice to mitigate dependency bloat. This is advocated by the official Maven guidelines [8]:

> *"Although transitive dependencies can implicitly include desired dependencies, it is a good practice to explicitly specify the dependencies you are directly using in your own source code. This best practice proves its value especially when the dependencies of your project changes their dependencies."*

However, in practice, this recommendation is not always followed by developers due to several reasons. For example, having all the dependencies in the POM produces a large build file that is more difficult to understand and maintain. It also induces a considerable maintenance burden to the projects, since dependencies will need to be updated one-by-one. According to the reviewer suggestion, we have added this topic to the discussion in Section 6.1 of the manuscript.

> **Comment 3.6** Bloated Dependencies: The authors should reconsider this example, as it confuses the reader a lot."

We have completely reworked the description of how the MDT and the DUT complement each other, sections 3.1 and 3.2. We have kept the original example, which grasps the richness of realistic dependency trees in the Maven ecosystem.

> **Comment 3.7**
> Bloated Dependencies: "The authors state that DepClean only leaves the dependencies that are currently needed, and removes directly declared unnecessary dependencies and excludes indirectly declared unnecessary dependencies. While this might work for release revisions, I'm not sure whether this is a good approach while implementing. Let's say I'm using dependency A in my code which depends on dependency B. I'm currently not using any code in A the makes B necessary. However, I may use such code in the future in my project or A may change in a way that now B is necessary. The authors need to describe also such cases as this might impact the applicability of the approach"

DepClean does not remove dependencies: it provides a report about bloated dependencies and creates a variant of the POM with the minimum set of dependencies to build the Maven project. DepClean does not perform any modifications to the source code, compiled bytecodes, or configurations files in the project. We have clarified this in the first paragraph of Section 3.3 in the manuscript.

Developers, who want to use DepClean, have to decide about how or where in their build pipeline they use the tool. In particular, the removal of bloated dependencies during release revisions is an appropriate method to use DepClean, which we have discussed with developers during our study. We have discussed other possibles methodologies, such as running DepClean as part of security audits or in the continuous integration pipeline to check only for direct bloated dependencies (see the second paragraph of Section 5.3.2). There are undoubtedly other possible usage methods and purposes. The systematic evaluation of these integration methods is out of the scope of this manuscript.

**Comment 3.8** Bloated Dependencies: "Additional remark to this: It would be interesting, how many of the removed/excluded dependencies will later be added again because they are then needed."

We agree with the reviewer; it would be very interesting to perform a more extensive longitudinal study along these lines. As a preliminary exploration of this topic, we have performed an initial analysis to determine which dependencies were added to the project after being removed or excluded. To do so, we have inspected the commit history of the POMs corresponding to the 30 projects used for RQ3 and RQ4.
We analyzed the output of the following shell commands:

```
git log -p -U0 pom.xml |
    grep "^+.*<artifactId>.*</artifactId>$\|
    ^-.*<artifactId>.*</artifactId>$" |
    awk "{$1=$1;print}" |
    sed "s/<artifactId>//" |
    sed "s/<\/artifactId>//"
```

Our preliminary results suggest that the declaration and removal of dependencies is very dynamic in large projects, and developers indeed add dependencies that were previously removed from the POM. A more extensive study on this topic involves the collection and analysis of a large set of build files and its git history, which is out of the scope of this work. The GitHub repository with our proof of concept tool to perform this kind of analysis is available at: https://github.com/castor-software/pomhist.

**Comment 3.9** Experimental Protocols: "In (1), the authors describe that they randomly sampled 14,699 Maven artifacts. First, I'd like to retrieve some more details than a plain reference about the methodology how these 14,699 were selected. At least, please, sketch the way how this selection has been performed.

Figure 1 compares the distribution of the original Maven Dependency Graph (MDG) dataset and the sample of 14,699 used in our manuscript. As we can observe from this figure, the 1st-Q and 3rd-Q are the same, which supports that our data is representative. This representativeness is achieved by sampling over the probability distribution of the number of direct dependencies per artifact in the MDG. According to the suggestion of the reviewer, we have added more details about the data selection methodology in Section 4.2.1 of the manuscript.

For more details about the implementations of this sampling technique, we refer the reviewer to the replication package: https://github.com/castor-software/depclean-experiments.

**Comment 3.10** Furthermore, the authors need to explain why the number of dependencies per artifact is a good measure for sampling the artifacts. Please, also describe why this number can be safely assumed as a good representative measure."

To our knowledge, the MDG is currently the largest representation of the Maven Central ecosystem publicly available for research. In this graph, we were unable to compute the total number of dependencies per artifact, i.e., direct and transitive dependencies, due to the high computational cost of calculating all the individual connections corresponding to more than 2 million nodes in the graph. Even assuming that we have the computing power, we cannot ensure than we count all the dependencies per artifact since many of them could be hosted in external repositories other than Maven Central, or unavailable during the Maven dependency resolution phase.

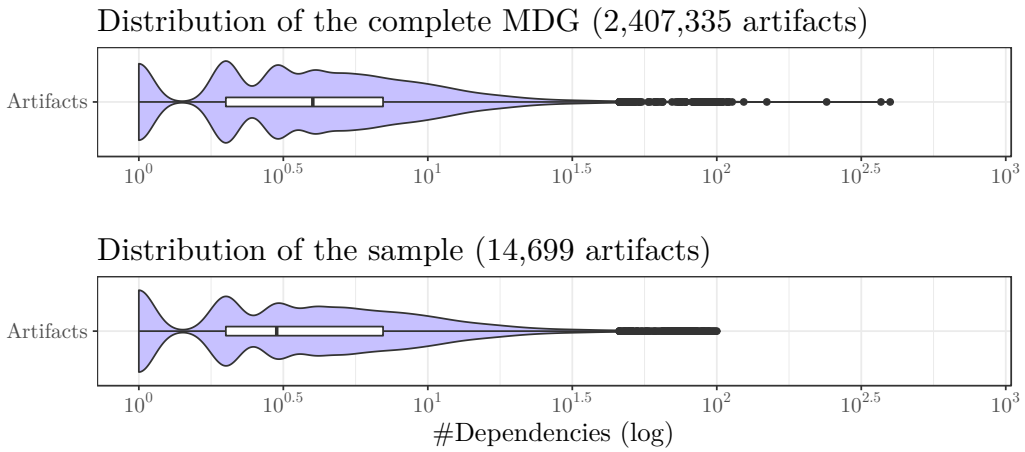


Figure 1: Distributions of the number of direct dependencies in the original MDG dataset of 2,407,335 artifacts, and the sample of 14,699 artifacts utilized in the manuscript.

Therefore, the number of direct dependencies per artifact, i.e., the number of outgoing edges per node, is the most natural criterion for sampling the artifact in the graph. We believe this is a good representative measure that fits the purpose and content of the paper because it reflects the diversity of code reuse in the Maven ecosystem. In our previous work [11], we found a positive correlation between the number direct dependencies and the number of transitive dependencies of artifacts in Maven Central. We have added more details regarding the sampling measure in Section 4.2.1 of the manuscript.

**Comment 3.11** Experimental Protocols: "The authors also describe additional criteria to further filter projects. The second criterion states that all subjects have different groupId and artifactId. Does that mean only one version of each artifact is considered (because different versions of a project usually share groupId and artifactId)? If yes, which one and why? If no, please rephrase this criterion as it might mislead the reader. Also the difference to the "Latest" criterion is then not fully clear."

We considered only one version of each artifact. When filtering artifacts to construct our dataset, we selected a single version per artifact belonging to the same *groupId* and *artifactId*. To select the version, we ordered the versions of each artifact according to its published date as available from Maven Central and selected the latest released version. This is explicitly stated in the bullet points of Section 4.2.1 of the manuscript.

**Comment 3.12** Experimental Protocols: "In the paragraph describing step (2), it is written that artifacts that ended up with an error were discarded. According to the number of used artifacts from the paragraph (1) which is 9,639, this never happened as the number of studied artifacts remains the same in this paragraph. Did errors happen? If yes, how many happened and could that impact the result?"

We thank the reviewer for this remark. The dataset of 9,639 artifacts is the result of applying the filtering criteria described above, and removing the artifacts that depend on libraries hosted in external repositories or that caused some errors when we download them. This happened for 131 artifacts in our sample. We have clarified this in Section 4.2.1 of the manuscript.

**Comment 3.13** Experimental Protocols: "The authors also present statistics about the studied subjects. They mention that it is interesting that the distribution of direct and transitive dependencies are notably different. I'd argue that this is as expected. Imagine that you have only single dependency that can have a large tail of transitive dependencies which again can have transitive dependencies and so on. Obviously, this will lead to a much higher number of transitive dependencies. The authors may rephrase this as, at least to me, this is not that interesting as it is sold."

We thank the reviewer for this remark. We have lowered down the tone of this observation in Section 4.2.1 of the manuscript, when describing the differences in the distribution of direct and transitive dependencies in our dataset.

**Comment 3.14** Experimental Protocols: "In (3), the authors describe how they decide whether a dependency is used. Why did the authors use such an approach? Wouldn't a simple slicing technique suffice to retrieve the same results? The authors should at least describe that there are different approaches possible, list some of them and describe the advantage of their approach over the others."

DepClean determines whether a dependency is used or bloated by constructing a static call graph of Java bytecode, to collect direct and indirect usages from the artifact to its dependencies. Indeed, the approach employed by DepClean can be seen as a form of slicing, where slices are computed by backtracking usages between the artifact and its dependencies. We have provided an extensive description of our approach in Section 3.3 of the manuscript.

**Comment 3.15** Experimental Protocols: "The authors then describe the dependency usage metrics. However, this paragraphs on page 15 confused me a lot. Many details are omitted here and as this might be a main contribution of the paper I'd recommend to describe the contents in more detail, for example answering the following questions:

- How is complexity of a Maven artifact measured? (Using only the height is not enough, also because some conclusions are made with this metric in the further sections)
- How can the single/multi module nature contribute to the results?
- Which reuse strategies exist and are used?
- How are the other dependency heights categorized?

Currently, this part reads as a vague itemization of some metrics that are somehow measured and I can currently not see how they contribute to answer the RQs.

We thank the reviewer for these valuable questions. We have refactored the description of the dependency usage metrics in the manuscript. We have established a mapping between the metrics described in section 4.2.1, and the corresponding figures in Section 5. We explained our criterion for measuring complexity based on the height of the dependency tree, as well as the purpose of measuring the difference between single and multi-module Maven projects.

**Comment 3.16** Experimental Protocols: "In the Protocol of the Qualitative Study, the authors select projects from GitHub. However, the selection criteria are given with no rationale. I'd expect at least some argumentation why the respective criterion is important. For example, why did they only select 30? Why 100 stars?"

We selected 30 open-source projects to conduct our qualitative analysis. The rationale of choosing this number of projects, is as follows. First, we need a reasonable number of notable open-source projects for which we can handle our pull requests and the discussion with their developers. Second, the projects should satisfy several conditions: 1) are popular in GitHub, 2) we can build them successfully, 2) contain dependencies, both used and bloated, 3) are actively maintained. We refer to these criteria in Section 3.2.2 of the manuscript.

We considered the number of stars of a repository as a direct measure of its popularity. Stars are the GitHub mechanism that provides a straightforward way for users to express their satisfaction with an open-source project. Researches in empirical software engineering have used starts as a measure to evaluate the popularity of projects [2, 12, 3]. We choose 100 as the minimum number of starts, which constitutes a standard threshold of popularity for a GitHub project.

**Comment 3.17** Experimental Results – 5.1 "In 5.1, it is stated that 75.1% of the dependencies are bloated. I'm not sure how to deal with this number. The authors emphasize that this is a huge number, which basically is correct, but if one looks at the detailed numbers, only 2.7% concern the direct dependencies which in my opinion are the most important because developer should intentionally declare their direct dependencies. While I see the argument of the authors to minimize long tail of dependencies that comes with the transitive closure of dependencies, I disagree that these are the important ones. The authors may want to improve their argumentation why the transitive ones are more important. Also, the numbers presented below Figure 10 (e.g. the 86.2%) are affected by this issue."

We agree with the reviewer: direct-bloated dependencies are easier to handle by developers, who only need to remove a few lines in the POM to get rid of them. However, notice that all the dependencies, whether direct or transitive will end up in the released binaries of the artifact. The effect of all the types of bloated dependencies are the same and consequently they are equally unwanted, and the difference lies in that the transitive-bloated dependencies are more challenging to remove and significantly more common. The pervasiveness of bloated-transitive dependencies that we found is the principal reason that explains its importance. Following the recommendation of the reviewer, we have made the distinction of the types of bloat explicit in the Abstract, Section 1 (Introduction), and Section 8 (Conclusions) of the manuscript.

**Comment 3.18** Experimental Results – 5.1 "The authors also report on the detailed numbers in this section. For example, the bloated-inherited dependencies are described as they would only happen in multi-module projects. As far as I know the multi-module nature does not imply inheritance of dependencies. To the best of my knowledge, inheritance in Maven-based systems is given through the declaration of a parent project, which is not mandatory, neither in single module projects, nor in multi module projects. The authors might clarify the text as this might be confusing to the reader."

Maven supports project aggregation in addition to project inheritance [7]. The *reactor* algorithm defines the specifications for handling multi-module projects. The multi-module Maven architecture provides two fundamental benefits for developers: 1) the ability to configure and customize the execution of the distinct Maven build phases for a set of specific modules, and 2) the facility of avoiding redundancy of POM configurations via inheritance, as in object-oriented programming, from sub-modules to a parent POM.

The declaration of a parent project is an optional design decision of the software architects. It does not necessarily imply the inheritance of dependencies. We have clarified this in Section 2.1 of the manuscript.

**Comment 3.19** Experimental Results – 5.1 "I appreciate that the authors also investigate the used dependencies. For example, I think the finding that many used dependencies are actually transitively declared dependencies is very interesting. The authors might highlight this finding more as this has the potential impact on how developers should declare their dependencies (= if you directly use an API, explicitly declare the dependency)."

We thank the reviewer for his/her interest in our findings regarding the usage of transitive dependencies. When developing this study, we were surprised by the diverse ways in which dependencies are used in practice. We have made and effort to present and discuss some interesting dependency usage cases with the help of Listings 2 and 3 in the manuscript.

**Comment 3.20** Experimental Results – 5.1 "Another important value that is missing in my opinion concerns the number of bloated transitive dependencies that gets erased when removing the 2.7% of directly declared bloated dependencies. How does this measure relate to the 75.1% of bloated transitive dependencies?"

We thank the reviewer for this remark. In RQ3, we have quantified the number of bloated-transitive dependencies that are removed as a result of removing bloated-direct dependencies. The numbers are reported in the fourth column of Table 4. The last row of this table summarized the acceptance rates: for the 25 bloated-direct dependencies removed, a total of 75 bloated dependencies were removed, 50 of them are bloated-transitive.

**Comment 3.21** Experimental Results – 5.2 "In this section, the authors describe the distribution of the type of dependencies over all of their studied artifacts (Figure 11). I wonder what the take-aways of this investigation are? Are some projects better than others because they declare more/less/other types of dependencies? An explanation of the impact of these numbers is needed."

We have added a paragraph to Section 5.2 of the manuscript that explains the practical impact of the numbers presented in Figure 11.

**Comment 3.22** Experimental Results – 5.2 "The same holds for Figure 12: This finding is anticipated. If there are more dependencies at all, I assume that there will also be more bloated dependencies. Please, put more emphasis on the practical applicability of this finding."

The assumption of the reviewer is correct: more transitive dependencies imply more bloat in general. However, this is not necessarily true for each artifact. The distribution of the number of transitive dependencies per artifact varies greatly (see Figure 10). There are artifacts with more than 1000 bloated-transitive, whereas most artifacts have between 2 and 41 (1st-Q and 3rd-Q), with a median of 11 bloated-transitive dependencies. Consequently, the correlation is not obvious for the reader (see the dispersion of artifacts in Figure 12).

Figure 12 ratifies the intuition of the reviewer and quantifies the impact of this phenomenon statistically. According to the suggestion of the reviewer, we have put more emphasis on the practical applicability of this finding in Section 5.2.1 of the manuscript.

**Comment 3.23** Experimental Results – 5.2 "The same holds for Figure 13: Please, give some examples of what one can learn from these numbers."

Figure 13 shows the relation between the height of the dependency tree and the number bloated dependencies. It shows a clear increasing trend of bloated-transitive dependencies as the height of the dependency tree increases. We have presented and described, in Section 5.2.1, the artifact `org.wso2.carbon.devicemgt:org.wso2.carbon.apimgt.handlers:3.0.192` as an example of this observation.

> **Comment 3.24** Experimental Results – 5.2 "In 5.2.2, the authors again try to find a relation between the way of constructing the project (multi-module vs. single module) and the number of bloated dependencies. Seeing the numbers, it seems that this has no impact on the number of dependencies per module. The authors should make the point clear here as I'm confused what this finding should convey. This holds for the whole RQ in my opinion. Ultimately, what the authors found is that if there are more dependencies declared, the chances of bloated dependencies increase. If the authors wanted to show a different effect, please describe this in more detail."

In Section 5.2.2, we compare the distributions of bloated and used dependencies between multi-module and single-module artifacts. The observation of the reviewer is correct: the Maven modular architecture has no impact on the number of dependencies per module, as compared with single-module artifacts. However, it does have an impact on the number of bloated dependencies (see Figures 14 and 15). We have clarified this point in Section 5.2.2, and referred to the need of tools and user interfaces to help developers manage their inherited dependencies.

This section is the result of our empirical observation when investigating the causes of dependency bloat. At the initial stage of this research, we did not know that the multi-module Maven architecture is one of the main causes of this type of bloat.

> **Comment 3.25** Experimental Results – 5.3 and 5.4 "Please, consider to refine the answers to RQ3 and RQ4 as it may be misleading to see 68+63=131 removed dependencies in the results sections and only the number 131 in the abstract. Same holds for the number of answered pull requests."

We have refined the answers to RQ3 and RQ4 in the manuscript as per this suggestion. We have also made this distinction explicit in the Abstract. Notice that we have updated the number of projects that answered our pull requests after the submission of the manuscript, according to the suggestion of Reviewer #2.

> **Comment 3.26** Experimental Results – 5.3 and 5.4 "The results of these RQs confirm my concern from above that the direct dependencies might be much more interesting than the transitive dependencies. The authors should take this into account and maybe revise some of the result descriptions in RQ1 and RQ2 based on these results."

We have emphasized on the importance of bloated-direct dependencies in Section 5.1, by adding a relevant sentence in the first paragraph of page 20 of the manuscript. This is also evidenced in the results presented in Section 5.3.

> **Comment 3.27** Discussion: "The authors mention the possible impact of their work on security aspects in the discussion. This has also already been mentioned in the introduction. While I agree with the authors that their approach can help to avoid security vulnerabilities, the whole paper does not investigate any security-related aspect it seems a little strange to again discuss this here. I'd expect at least some more details on security relevant removed dependencies to justify the link."

We have removed from the manuscript the paragraph that discusses the motivations of performing dependency analysis for security. We still want to mention to the reviewer that we have confirmed with our industrial partners the practical relevance of the study of security aspects for better dependency management.

**Minor remarks**

All the typos pointed out by the reviewer have been corrected in the resubmitted version of our manuscript.

# References


[1] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The Maven Dependency Graph: a Temporal Graph-based Representation of Maven Central. In *16th International Conference on Mining Software Repositories (MSR)*, Montreal, Canada, 2019. IEEE/ACM.

[2] Hudson Borges and Marco Tulio Valente. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129, 2018.

[3] Junxiao Han, Shuiguang Deng, Xin Xia, Dongjing Wang, and Jianwei Yin. Characterization and Prediction of Popular Projects on GitHub. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 21–26. IEEE, 2019.

[4] Joseph Hejderup. In Dependencies We Trust: How Vulnerable are Dependencies in Software Modules? 2015.

[5] Joseph Hejderup. PRÄZI: From Package-based to Precise Call-based Dependency Network Analyses. 2018.

[6] Gradle User Manual. Understanding Dependency Resolution. https://docs.gradle.org/current/userguide/dependency_resolution.html, 2020. [Online; accessed 30-April-2020].

[7] Apache Maven Project. Guide to Working with Multiple Modules. http://maven.apache.org/guides/mini/guide-multiple-modules.html, 2020. [Online; accessed 30-April-2020].

[8] Apache Maven Project. Introduction to the Dependency Mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html, 2020. [Online; accessed 30-April-2020].

[9] Apache Maven Project. The Maven Dependency Analyzer. http://maven.apache.org/shared/maven-dependency-analyzer, 2020. [Online; accessed 30-April-2020].

[10] The Apache Ant Project. Apache Ivy. https://ant.apache.org/ivy, 2020. [Online; accessed 30-April-2020].

[11] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in maven central. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR'19, pages 333 – 343. IEEE Press, 2019.

[12] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. Association for Computing Machinery.