Empirical Software Engineering (2021) 26: 45
https://doi.org/10.1007/s10664-020-09914-8

A comprehensive study of bloated dependencies in the Maven ecosystem

César Soto-Valero[1] · Nicolas Harrand[1] · Martin Monperrus[1] · Benoit Baudry[1]

Accepted: 23 September 2020 / Published online: 25 March 2021

Abstract
Build automation tools and package managers have a profound influence on software development. They facilitate the reuse of third-party libraries, support a clear separation between the application's code and its external dependencies, and automate several software development tasks. However, the wide adoption of these tools introduces new challenges related to dependency management. In this paper, we propose an original study of one such challenge: the emergence of bloated dependencies. Bloated dependencies are libraries that are packaged with the application's compiled code but that are actually not necessary to build and run the application. They artificially grow the size of the built binary and increase maintenance effort. We propose DepClean, a tool to determine the presence of bloated dependencies in Maven artifacts. We analyze 9,639 Java artifacts hosted on Maven Central, which include a total of 723,444 dependency relationships. Our key result is as follows: 2.7% of the dependencies directly declared are bloated, 15.4% of the inherited dependencies are bloated, and 57% of the transitive dependencies of the studied artifacts are bloated. In other words, it is feasible to reduce the number of dependencies of Maven artifacts to 1/4 of its current count. Our qualitative assessment with 30 notable open-source projects indicates that developers pay attention to their dependencies when they are notified of the problem. They are willing to remove bloated dependencies: 21/26 answered pull requests were accepted and merged by developers, removing 140 dependencies in total: 75 direct and 65 transitive.

Keywords Dependency management · Software reuse · Debloating · Program analysis

1 Introduction

Software reuse, a long time advocated software engineering practice (Naur and Randell 1969; Krueger 1992), has boomed in the last years thanks to the widespread adoption of build automation and package managers (Cox 2019; Soto-Valero et al. 2019). Package managers provide both a large pool of reusable packages, a.k.a. libraries, and systematic ways to

Communicated by: Gabriele Bavota

✉ César Soto-Valero
cesarsv@kth.se

[1] KTH Royal Institute of Technology, Stockholm, Sweden

declare what are the packages on which an application depends. Examples of such package management systems include Maven for Java, npm for Node.js, or Cargo for Rust. Build tools automatically fetch all the packages that are needed to compile, test, and deploy an application.

Package managers boost software reuse by creating a clear separation between the application and its third-party dependencies. Meanwhile, they introduce new challenges for the developers of software applications, who now need to manage those third-party dependencies (Cox 2019) to avoid entering into the so-called "dependency hell". These challenges relate to ensuring high quality dependencies (Salza et al. 2019), keeping the dependencies up-to-date (Bavota et al. 2015), or making sure that heterogeneous licenses are compatible (Wu et al. 2017).

Our work focuses on one specific challenge of dependency management: the existence of *bloated dependencies*. This refers to packages that are declared as dependencies for an application, but that are actually not necessary to build or run it. The major issue with bloated dependencies is that the final deployed binary file includes more code than necessary: an artificially large binary is an issue when the application is sent over the network (e.g., web applications) or it is deployed on small devices (e.g., embedded systems). Bloated dependencies could also embed vulnerable code that can be exploited, while being actually useless for the application (Gkortzis et al. 2019). Overall, bloated dependencies needlessly increase the difficulty of managing and evolving software applications.

We propose a novel, unique, and large scale analysis of bloated dependencies. So far, research on bloated dependencies has been only touched with a study of copy-pasted dependency declarations by McIntosh et al. (2014), and a study of unused dependencies in the Rust ecosystem by Hejderup et al. (2018). Our previous work gives preliminary results on this topic in the context of Java (Harrand et al. 2019). Yet, there is no systematic analysis of the presence of bloated dependencies nor about the importance of this problem for developers in the Java ecosystem.

Our work focuses on Maven, the most popular package manager and automatic build system for Java and languages that compile to the JVM. In Maven, developers declare dependencies in a specific file, called the POM file. In order to analyze thousands of artifacts on Maven Central, the largest repository of Java artifacts, manual analysis is not a feasible solution. To overcome this problem, we have developed DepClean, a tool that performs an automatic analysis of dependency usage in Maven projects. Given an application and its POM file, DepClean collects the complete dependency tree (the list of dependencies declared in the POM, as well as the transitive dependencies) and analyzes the bytecode of the artifact and all its dependencies to determine the presence of bloated dependencies. Finally, DepClean generates a variant of the POM in which bloated dependencies are removed.

Armed with DepClean, we structured our analysis of bloated dependencies in two parts. First, we automatically analysed 9,639 artifacts and their 723,444 dependencies. We found that 75.1% of these dependencies are bloated. We identify transitive dependencies and the complexities of dependency management in multi-module projects as the primary causes of bloat. Second, we performed a user study involving 30 artifacts, for which the code is available as open-source on GitHub and which are actively maintained. For each project, we used DepClean to generate a POM file without bloated dependencies and submitted the changes as a pull request to the project. Notably, our work yielded 21 merged pull requests by open-source developers and 140 bloated dependencies were removed.

To summarize, this paper makes the following contributions:

- A comprehensive study of bloated dependencies in the context of the Maven package manager. We are the first to quantify the magnitude of bloat on a large scale (9,639 Maven artifacts) showing that 75.1% of dependencies are bloated.
- A tool called DepClean to automatically analyze and remove bloated dependencies in Java applications packaged with Maven. DepClean can be used in future research on package management as well as by practitioners.
- A qualitative assessment of the opinion of developers regarding bloated dependencies. Through the submission of pull requests to notable open-source projects, we show that developers care about removing dependency bloat: 21/26 of answered pull requests have been merged, removing 140 bloated dependencies.

The remainder of this paper is structured as follows. Section 2 introduces the key concepts about dependency management with Maven and presents an illustrative example. Section 3 introduces the new terminology and describes the implementation of DepClean. Section 4 presents the research questions that drive our study, as well as the methodology followed. Section 5 covers our experimental results for each research question. Sections 6 and 7 provide a comprehensive discussion of the results obtained and present the threats to the validity of our study. Section 8 concludes this paper and provides future research directions.

2 Background

We provide an overview of the Maven package management system and of the essential terminology. We illustrate these concepts with a concrete example.

2.1 Maven Dependency Management Terminology

Maven is a popular package manager and build automation tool for Java projects and other languages that compile to the JVM (e.g., Scala, Kotlin, Groovy, Clojure, or JRuby). Maven relies on a specific build file, known as the POM (acronym for "Project Object Model"), where developers specify information about the project, its dependencies and its build process. POM files can inherit from a base POM, known as the Maven parent POM. The inheritance and declaration of dependencies is a design decision of developers.

Maven Project A Maven project includes source code files and build files. It can be a *single-module*, or a *multi-module* project. The former has a single POM file, which includes all the dependencies and build instructions to produce a single artifact (JAR file). The latter allows to separately build multiple artifacts in a certain order through a so-called aggregator POM. In multi-module projects, developers can define a parent POM that specifies the dependencies used by all the modules.

Maven Artifact We refer to *artifacts* as compiled Maven projects that have been deployed to a binary code repository. A Maven artifact is typically a JAR file that is uniquely identified with a triplet (G:A:V), G, the *groupId*, identifies the organization that develops the artifact, A, the *artifactId*, is the name of the artifact, and V corresponds to its *version*. Maven Central is the most popular public repository to host Maven artifacts.

Dependency Resolution Maven resolves dependencies in two steps: (1) based on the POM file(s), it determines the set of required dependencies, and (2) it fetches dependencies that are not present locally from external repositories such as Maven Central. Maven constructs a *dependency tree*, that captures all dependencies and their relationships. Given one artifact, we distinguish between three types of Maven dependencies: *direct* dependencies that are explicitly declared in the POM file; *inherited* dependencies, which are declared in the parent POM; and *transitive* dependencies obtained from the transitive closure of direct and inherited dependencies. Dependency version management is a key feature of the dependency resolution, which Maven handles with a specific *dependency mediation algorithm*.[1]

2.2 A Brief Journey in the Dependencies of the JXLS Library

We illustrate the concepts introduced previously with one concrete example: JXLS,[2] an open source Java library for generating Excel reports. It is implemented as a multi-module Maven project with a parent POM in `jxls-project`, and three modules: `jxls`, `jxls-examples`, and `jxls-poi`.

Listing 1 shows an excerpt of the POM file of the `jxls-poi` module, version 1.0.15. It declares `jxls-project` as its parent POM (lines 1 – 5) and a direct dependency towards the `poi` Apache library (lines 10–14). Figure 1 depicts an excerpt of its Maven dependency tree (we do not show testing dependencies here, such as JUnit, to make the figure more readable). Nodes in blue, pink, and yellow represent direct, inherited, and transitive dependencies, respectively, for the `jxls-poi` artifact (as reported by the `dependency:tree` Maven plugin).

Listing 1 Excerpt of the POM file corresponding to the module `jxls-poi` of the multi-module Maven project JXLS

```
1  <parent>
2     <groupId>org.jxls</groupId>
3     <artifactId>jxls-project</artifactId>
4  <version>2.6.0</version>
5  </parent>
6  <artifactId>jxls-poi</artifactId>
7  <packaging>jar</packaging>
8  <version>1.0.15</version>
9  <dependencies>
10    <dependency>
11       <groupId>org.apache.poi</groupId>
12       <artifactId>poi</artifactId>
13       <version>3.17</version>
14    </dependency>
15    ...
16 </dependencies>
```

The library `jcl-over-slf4j` declares a dependency towards `slf4j-api`, version 1.7.12, which is omitted by Maven since it is already added from the `jxls-project` parent POM. On the other hand, JXLS declares dependencies to version 1.7.26 of `jcl-over-slf4j` and `slf4j-api`, but the lower version 1.7.12 was

[1] https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.
[2] http://jxls.sourceforge.net

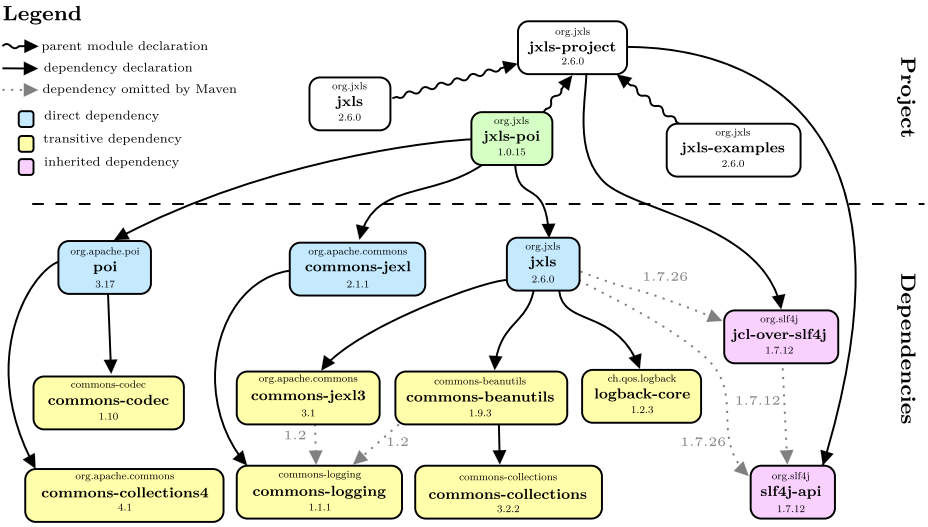


Fig. 1 Excerpt of the dependency tree of the multi-module Maven project JXLS (dependencies used for testing are not shown for the sake of simplicity)

chosen over it since it is nearer to the root in the dependency tree and, by default, Maven resolves version conflicts with a nearest-wins strategy. Once Maven finishes the dependency resolution, the classpath of `jxls-poi` includes the following artifacts: `poi`, `commons-codec`, `commons-collections4`, `commons-jexl`, `commons-logging`, `jxls`, `commons-jex13`, `commons-beanutils`, `commons-collections`, `logback-core`, `jcl-over-slf4j`, and `slf4j-api`. The goal of our work is to determine if all the artifacts in the classpath of Maven projects such as `jxls-poi` are actually needed to build and run those projects.

3 Bloated Dependencies

In this section, we introduce the idea of bloated dependency, which is the fundamental concept presented and studied in the rest of this paper. We describe our methodology to study bloated dependencies, as well as our tools to automatically detect and remove them from Maven artifacts.

Dependencies among Maven artifacts form a graph, according to the information declared in their POMs. This graph has been introduced in our previous work about the Maven Dependency Graph (MDG) (Benelallam et al. 2019). The MDG is defined as follows:

Definition 1 (Maven Dependency Graph) The MDG is a vertex-labelled graph, where vertices are Maven artifacts (uniquely identified by their $G{:}A{:}V$ coordinates), and edges represent dependency relationships among them. Formally, the MDG is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where:

- $\mathcal{V}$ is the set of artifacts in the Maven Central repository
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represent the set of directed edges that determine dependency relationships between each artifact $v \in \mathcal{V}$ and its dependencies

3.1 Novel Concepts

Each artifact in the MDG has its own Maven Dependency Tree (MDT), which is the transitive closure of all the dependencies needed to build the artifact, as resolved by Maven.

Definition 2 (Maven Dependency Tree) The MDT of an artifact $v \in \mathcal{V}$ is a directed acyclic graph of artifacts, with v as the root node, and a set of edges $\mathcal{E}$ representing dependency relationships between them.

In this work, we introduce the novel concept of *bloated dependency* as follows:

Definition 3 (Bloated Dependency) An artifact p is said to have a bloated dependency relationship $\varepsilon_b \in \mathcal{E}$ if there is a path in its MDT, between p and any dependency d of p, such that none of the elements in the API of d are used, directly or indirectly, by p.

To reason about the bloated dependencies of an artifact, we introduce a new data structure, called the Dependency Usage Tree (DUT) as follows.

Definition 4 (Dependency Usage Tree) The DUT of an artifact a, defined as $\text{DUT}_a = (\mathcal{V}, \mathcal{E}, \nabla)$, is a tree, whose nodes are the same as the Maven Dependency for a and which edges are all of the (a, a_i), for all nodes $a_i \in \text{DUT}_a$. A labeling function ∇ assigns each edge one of the following six dependency usage types: $\nabla : \mathcal{E} \rightarrow \{\text{ud, ui, ut, bd, bi, bt}\}$ such that:

$$\nabla(\langle p, d \rangle) = \begin{cases} \text{ud}, & \text{if } d \text{ is used and it is directly declared by } p \\ \text{ui}, & \text{if } d \text{ is used and it is inherited from a parent of } p \\ \text{ut}, & \text{if } d \text{ is used and it is resolved transitively by } p \\ \text{bd}, & \text{if } d \text{ is bloated and it is directly declared by } p \\ \text{bi}, & \text{if } d \text{ is bloated and it is inherited from a parent of } p \\ \text{bt}, & \text{if } d \text{ is bloated and it is resolved transitively by } p \end{cases}$$

It is to be noted that, in the case of transitive dependencies, ∇ assigns the bt label to the relationship (a, a_i) if and only if two conditions hold: 1) a does not use any member of a_i, and 2) none of the artifacts in the tree need to use a_i to fulfill the requirements of a.

Given a Maven artifact a we build both the MDT_a and the DUT_a. Both trees include exactly the same set of nodes, but the edges are different. In the MDT_a, an edge (a_1, a_2) exists when the POM of a_1 declares a dependency towards a_2. In the DUT_a, all edges start from a, and an edge (a, a_1) means that a_1 is an artifact in the MDT_a. In the case a_1 is not

a direct dependency of a, then the edge (a, a_1) does not exist in the MDT_a, yet we need to model it, since it is the relation (a, a_1) that can be bloated or used.

3.2 Example

Figure 2 illustrates the dependency usage tree for the example presented in Fig. 1. Analyzing the bytecode of `jxls-poi`, we find no references to any API member of the direct dependencies `commons-jexl` (explicitly declared in the POM) and `sl4j` (inherited from its parent POM). Therefore, these dependency relationships are labelled as bloated-direct (bd) and bloated-inherited (bi) dependencies, respectively.

Now let us consider the dependency to `commons-codec`. In the MDT of `jxls-poi` (cf. Fig. 1), we observe that `commons-codec` is a transitive dependency of `jxls-poi`, through `poi`. From the perspective of bloat, what we want to know is the following: is `commons-codec` necessary to build and run `jxls-poi`? Therefore, we are interested in the relationship between `jxls-poi` and `commons-codec`, which we model in the DUT of `jxls-poi` (cf. Fig. 2). To answer the question of usage we need two analyses. First, an analysis of the bytecode of `jxls-poi` reveals that it does not use `commons-codec` directly. Second, we observe that `jxls-poi` uses some members of `poi`, and that these members of `poi` do not use `commons-codec`. So, we conclude that the relationship between `jxls-poi` and `commons-codec` is bloated, and the corresponding edge is labelled bt. It is important to note that a bloated transitive dependency relationship between `jxls-poi` and `commons-codec` does not mean that `commons-codec` is bloated for `poi`, but only for the subpart of `poi` that is necessary for `jxls-poi`. Table 1 summarizes the labelling of all the dependency relationships of `jxls-poi`.

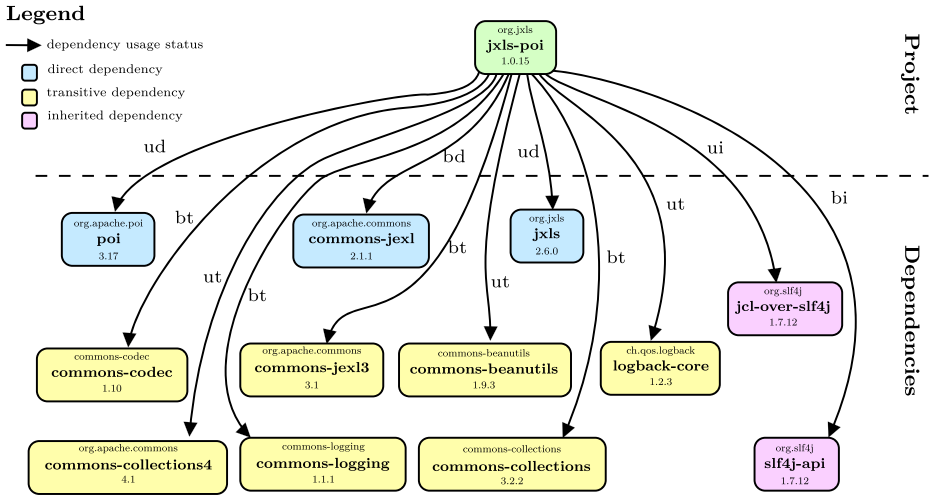


Fig. 2 Dependency Usage Tree (DUT) for the example presented in Fig. 1. Edges are labelled according to Definition 4 to reflect the usage status between `jxls-poi` and each one of its dependencies

Table 1 Contingency table of the different types of dependency relationships studied in this work for the example presented in Fig. 2

	Used	Bloated
Direct	poi, jxls	commons-jexl
Inherited	jcl-over-sl4j	sl4j-api
Transitive	commons-beanutils,logback-core, commons-collections4	commons-logging, commons-collections, commons-codec, commons-jexl3

3.3 DepClean: A Tool for Detecting and Removing Bloated Dependencies

For our study, we design and implement a specific tool called DepClean. An overview of DepClean is shown in Fig. 3, it works as follows. It receives as inputs a built Maven project and a repository of artifacts, then it extracts the dependency tree of

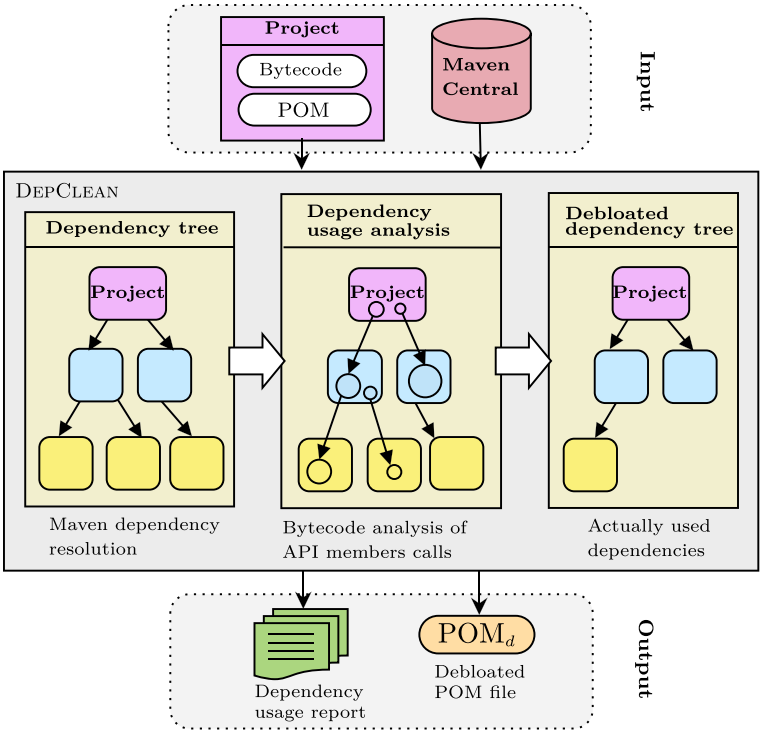


Fig. 3 Overview of the tool DepClean to detect and remove bloated dependencies in Maven projects. Rounded squares represent artifacts, circles inside the artifacts are API members, arrows between API members represents bytecode calls between artifacts, arrows between artifacts represent dependency relationships between them

the projects and constructs a DUT to identify the set of dependencies that are actually used by the project. DepClean has two outputs: (1) it returns a report with the usage status of all types of dependencies, and (2) it produces an alternative version of the POM file (POM_d) with all the bloated dependencies removed (i.e., the XML node of the bloated dependency is removed). DepClean does not perform any modifications to the source code, bytecode, or configurations files in the project under consideration.

Algorithm 1 details the main procedure of DepClean. The algorithm receives as input a Maven artifact p that includes a set of dependencies in its dependency tree, denoted as DT, and returns a report of the usage status of its dependencies and a debloated version of its POM. Notice that DepClean computes two transitive closures: over the Maven dependency tree (line 2) and over the call graph of API members (line 3).

Algorithm 1 Main procedure to detect and remove bloated dependencies.

Input: A Maven artifact p.
Output: A report of the bloated dependencies of p and a clean POM file f of p without bloated dependencies.

```
1  f ← copyPOM(p);
2  DT ← getDependencyTree(p);
3  UD ← getUsedDependencies(p, DT); // refer to Algorithm 2
4  foreach d ∈ DT do
5      if d ∈ UD then
6          continue; // do nothing, the dependency is actually
                used by p
7      end
8      if isDeclared(d, f) then
9          report d as bloated-direct and remove(d, f);
10     else
11         report d as bloated-inherited or bloated-transitive, and exclude(d, f);
12     end
13 end
14 return f;
```

The algorithm first copies the POM file of p, resolving all its direct and transitive dependencies locally, and obtaining the dependency tree (lines 1 and 2). If p is a module of a multi-module project, then all the dependencies declared in its parent POM are included as direct dependencies of p. Then, the algorithm proceeds to construct a set with the dependencies that are actually used by p (line 3).

Algorithm 2 explains the bytecode analysis. The detection component statically analyzes the bytecode of p and all its dependencies to check which API members are being referenced by the artifact, either directly or indirectly. Notice that it behaves differently if the included artifact is a direct, inherited, or a transitive dependency. If none of the API members of a dependency $d \in DT$ are called, even indirectly via transitive dependencies, then d is considered to be bloated, and we proceed to remove it.

Algorithm 2 Procedure to obtain all the dependencies used, directly or indirectly, by a Maven artifact.

Input: A Maven artifact p and its dependency tree DT.
Output: A set of dependencies UD actually used by p.

```
1  UD ← ∅;
2  foreach d ∈ DT do
3      if (p, d) is a direct or inherited dependency then
4          part_d ← extractMembers(p, d); // extract the subpart of d
               used by p
5          if part_d ≠ ∅ then
6              add(d, UD);
7          else
8              continue;
9          end
10     else
11         find the path l = [p, ..., d] connecting p and d in DT;
12         a ← p;
13         foreach b ∈ l_{2,...,n} do
14             part_b ← extractMembers(a, b);
15             if part_b = ∅ then
16                 break;
17             else
18                 a ← part_b;
19             end
20             if b = d then
21                 add(d, UD);
22             end
23         end
24     end
25 end
```

In Maven, we remove bloated dependencies in two different ways: (1) if the bloated dependency is explicitly declared in the POM, then we remove its declaration clause directly (line 9 in Algorithm 1), or (2) if the bloated dependency is inherited from a parent POM or induced transitively, then we excluded it in the POM (line 11 in Algorithm 1). This exclusion consists in adding an `<exclusion>` clause inside a direct dependency declaration, with the *groupId* and *artifactId* of the transitive dependency to be excluded. Excluded dependencies are not added to the classpath of the artifact by way of the dependency in which the exclusion was declared.

DepClean is implemented in Java as a Maven plugin that extends the `maven-dependency-analyzer`[3] tool, which is actively maintained by the Maven team and officially supported by the Apache Software Foundation. For the construction of the dependency tree, DepClean relies on the `copy-dependencies` and `tree` goals of the `maven-dependency-plugin`. Internally, DepClean relies on the ASM[4] library

[3] http://maven.apache.org/shared/maven-dependency-analyzer
[4] https://asm.ow2.io

to visit all the `.class` files of the compiled projects in order to register bytecode calls towards classes, methods, fields, and annotations among Maven artifacts and their dependencies. For example, it captures all the dynamic invocations created from class literals by parsing the bytecodes in the constant pool of the classes. DEPCLEAN defines a customized parser that reads entries in the constant pool of the `.class` files directly, in case it contains special references that ASM does not support. This allows the plugin to statically capture reflection calls that are based on string literals and concatenations. Compared to `maven-dependency-analyzer`, DEPCLEAN adds the unique features of detecting transitive and inherited bloated dependencies, and to produce a debloated version of the POM file. DEPCLEAN is open-source and reusable from Maven Central, the source code is available at https://github.com/castor-software/depclean.

4 Experimental Methodology

In this section, we present the research questions that articulate our study. We also describe the experimental protocols used to select and analyze Maven artifacts for an assessment of the impact of bloated dependencies in this ecosystem.

4.1 Research Questions

Our investigation of bloated dependencies in the Maven ecosystem is composed of four research questions grouped in two parts. In the first part, we perform a large scale quantitative study to answer the following research questions:

- **RQ1**: *How frequently do bloated dependencies occur?* With this research question, we aim at quantifying the amount of bloated dependencies among 9,639 Maven artifacts. We measure direct, inherited and transitive dependencies to provide an in-depth assessment of the dependency bloat in the Maven ecosystem.
- **RQ2**: *How do the reuse practices affect bloated dependencies?* In this research question, we analyze bloated dependencies with respect to two distinctive aspects of reuse in the Maven ecosystem: the additional complexity of the Maven dependency tree caused by transitive dependencies, and the choice of a multi-module architecture.

The second part of our study focuses on 30 notable Maven projects and presents the qualitative feedback about how developers react to bloated dependencies, and to the solutions provided by DEPCLEAN. It is guided by the following research questions:

- **RQ3**: *To what extent are developers willing to remove bloated-direct dependencies?* Direct dependencies are those that are explicitly declared in the POM. Hence, those dependencies are easy to remove since it only requires the modification of a POM that developers can easily change. In this research question, we use DEPCLEAN to detect and fix bloated-direct dependencies. Then, we communicate the results to the developers. We report on their feedback.
- **RQ4**: *To what extent are developers willing to exclude bloated-transitive dependencies?* Transitive dependencies are those not explicitly declared in the POM but induced from other dependencies. We exchange with developers about such cases. This gives unique insights about how developers react to excluding transitive dependencies from their projects.

4.2 Experimental Protocols

4.2.1 Protocol of the Quantitative Study (RQ1 & RQ2)

Figure 4 shows our process to build a dataset of Maven artifacts in order to answer RQ1 and RQ2. Steps ① and ② focus on the collection of a representative set, according to the number of direct dependencies, of Maven artifacts: we sample our study subjects from the whole MDG, then we resolve the dependencies of each study subject. In steps ③ and ④, we analyze dependency usages with DEPCLEAN and compute the set of metrics to answer RQ1 and RQ2.

Filter Artifacts In the first step, we leverage the Maven Dependency Graph (MDG) from previous research (Benelallam et al. 2019), a graph database that captures the complete dependency relationships between artifacts in Maven Central at a given point in time. Figure 5a shows the distribution of the number of direct dependencies of the artifacts with at least one direct dependency in the MDG. The number of direct dependencies is a representative measure that reflects the initial intentions of developers with respect to code reuse. We select a representative sample that includes 14,699 Maven artifacts (Fig. 5b). Representativeness is achieved by sampling over the probability distribution of the number of direct dependencies per artifact in the MDG, per the recommendation of Shull (2007, Chapter 8.3.1). From the sampled artifacts, we select as study subjects all the artifacts that meet the following additional criteria:

- Public API: The subjects must contain at least one `.class` file with one or more public methods, i.e., can be reused via external calls to their API.
- Diverse: The subjects all have different *groupId* and *artifactId*, i.e., they belong to different Maven projects.

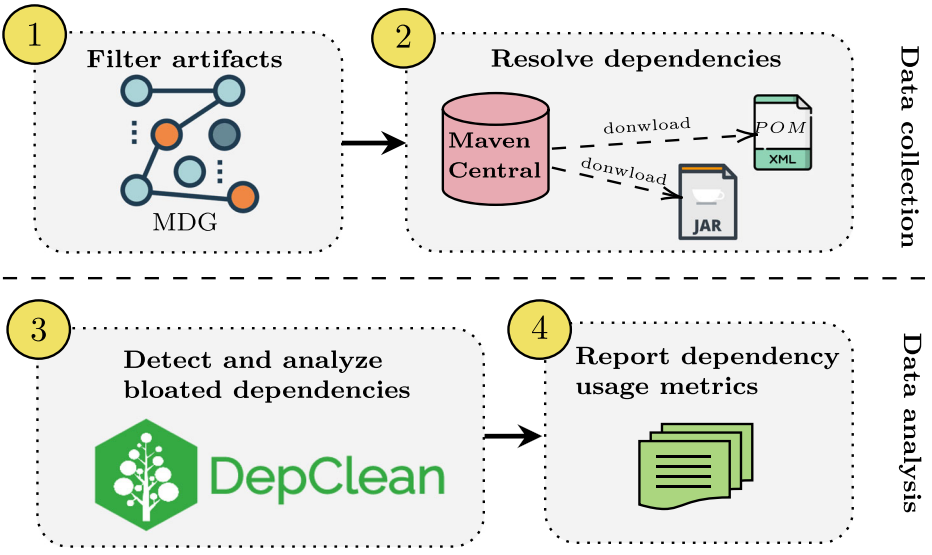


Fig. 4 Experimental framework used to collect artifacts and analyze bloated dependencies in the Maven ecosystem

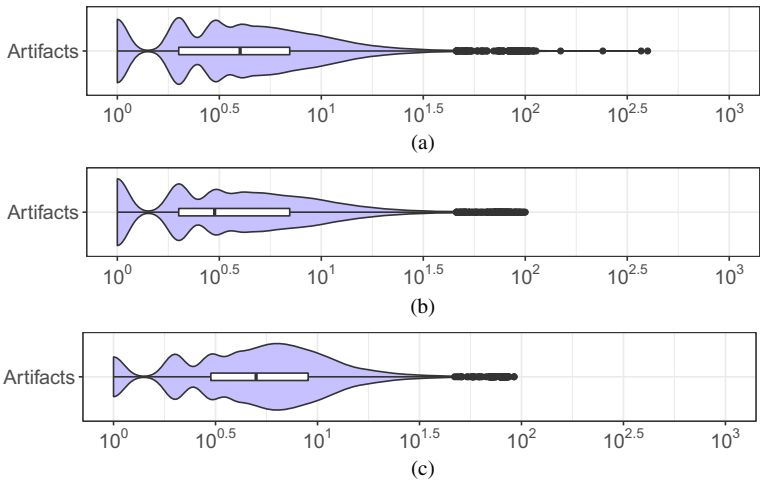


Fig. 5 Distribution of the number of direct dependencies of the artifacts at the different stages of the data filtering process

- Reused: The subjects are used by at least one client via direct declaration.
- Complex: The subjects have at least one direct dependency with *compile* scope, i.e., we can analyze the dependency tree and the reused artifacts.
- Latest: The subjects are the latest released version at the time of the experiment (October, 2019).

After this systematic selection procedure, we obtain a dataset of 9,770 Maven artifacts. The density of artifacts with a number of direct dependencies in the range [3, 9] in our dataset (Fig. 5c) is higher than in the MDG (Fig. 5a). This is a direct consequence of our selection criteria where artifacts must have at least one direct dependency with *compile* scope. This filter removes artifacts that contain only dependencies that are not shipped in the JAR of the artifact (e.g., *test* dependencies). Therefore, the 9,770 artifacts used as study subjects are representative of the artifacts in Maven Central that include third-party dependencies in the JAR.

Resolve Dependencies In the second step, we download the binaries of all the selected artifacts and their POMs from Maven Central and we resolve all their direct and transitive dependencies to a local repository. To ensure the consistency of our analysis, we discard the artifacts that depend on libraries hosted in external repositories. In case of any other error when downloading some dependency, we exclude the artifact from our analysis. This occurred for a total of 131 artifacts in the dataset obtained in the first step.

Table 2 shows the descriptive statistics about the 9,639 artifacts in our final dataset for RQ1 and RQ2. The dataset includes 44,488 direct, 180,693 inherited, and 498,263 transitive dependency relationships (723,444 in total). We report about their dependencies with *compile* scope, since those dependencies are necessary to build the artifacts. Columns #C, #M, and #F give the distribution of the number of classes, methods, and fields per artifact (we count both the public and private API members). The size of artifacts varies, from small artifacts with one single class (e.g., `org.elasticsearch.client:transport:6.2.4`), to large libraries with thousands of classes (e.g., `org.apache.hive:hive-exec:3.1.0`). In total, we analyze

Table 2 Descriptive statistics of the 9,639 Maven artifacts selected to conduct our quantitative study of bloated dependencies (RQ1 & RQ2)

	API Members			Dependencies		
	#C	#M	#F	#D	#I	#T
Min.	1	1	0	1	0	0
1st-Q	10	63	21	2	0	6
Median	32	231	75	4	2	20
3rd-Q	111	891	279	7	18	59
Max.	47,241	435,695	129,441	148	453	1,776
Total	2,397,879	22,633,743	6,510,584	44,488	180,693	498,263

the bytecode of more than 30 millions of API members. Columns #D, #I, and #T account for the distributions of direct, inherited, and transitive dependencies, respectively. `com.bbossgroups.pdp:pdp-system:5.0.3.9` is the artifact with the largest number of declared dependencies in our dataset, with 148 dependency declarations in its POM file, while `be.atbash.json:octopus-accessors-smart:0.9.1` has the maximum number of transitive dependencies: 1,776. The distributions of direct and transitive dependencies are notably different: typically the number of transitive dependencies is an order of magnitude larger than direct dependencies, with means of 20 and 4, respectively.

Dependency Usage Analysis This is the first step to answer RQ1 and RQ2: collect the status of all dependency relationships for each artifact in our dataset. For each artifact, we first unpack its JAR file, as well as its dependencies. Then, for each JAR file, we analyze all the bytecode calls to API class members using DepClean. This provides a Dependency Usage Tree (DUT) for each artifact, on which each dependency relationship is labeled with one of the six categories as we illustrated in Table 1: bloated-direct (bd), bloated-inherited (bi), bloated-transitive (bt), used-direct (ud), used-inherited (ui), or used-transitive (ut).

Collect Dependency Usage Metrics This last step consists of collecting a set of metrics about the global presence of bloated dependencies. We define our analysis metrics with the goals of studying 1) the dependency usage relationships in the DUT (RQ1), and 2) the complexity resulting from the adoption of the multi-module Maven architecture (RQ2).

In RQ1, we analyze our dataset as a whole, looking at the usage status of dependency relationships from two perspectives:

- *Global distribution of dependency usage.* This is the normalized distribution of each category of dependency usage, over each dependency relationship of each of the 9,639 artifacts in our dataset.
- *Distribution of dependency usage type, per artifact.* For each of the six types of dependency usage, we compute the normalized ratio over the total number of dependency relationships for each artifact in our dataset.

In RQ2, we analyze how the specific reuse strategies of Maven relate to the presence of bloated dependencies. First, we use the number of transitive dependencies and the height of the dependency tree as a measure of complexity. The former measure is guided by the fact that transitive dependencies are more difficult to handle by developers; the latter measure is

guided by the idea that dependencies that are deeper in the dependency tree are more likely to be bloated. We use the following metrics:

- *Bloated dependencies w.r.t. the number of transitive dependencies.* For each artifact that has at least one transitive dependency, we determine the relation between the ratio of transitive dependencies and the ratio of bloated dependencies.
- *Bloated-transitive dependencies w.r.t. to the height of the dependency tree.* Given an artifact and its Maven dependency tree, the height of the tree is the longest path between the root and its leaves. To compute this metric, we group our artifacts according to the height of their tree. The maximum dependency tree height that we observed is 14. However, there are only 152 artifacts with a tree higher than 9. Therefore, we group all artifacts with height ≥ 9. For each subset of artifact with the same height, we compute the size of the subset and the distribution of bloated-transitive dependencies of each artifact in the subset.

Second, we distinguish the presence of bloated dependencies between single and multi-module Maven projects, according to the following metrics:

- *Global distribution of dependency usage in a single or multi-module project.* We present two plots that measure the distribution of each type of dependency usage in the set of single and multi-module projects. It is to be noted that the plot for single-module projects does not include bloated-inherited (bi) and used-inherited (ui) dependencies since they have no inherited dependencies.
- *Distribution of dependency usage type, per artifact, in a single or multi-module project.* We present two plots that provide six distributions each: the distribution of each type of dependency usage type for artifacts that are in a single-module or multi-module project.

4.2.2 Protocol of the Qualitative Study (RQ3 & RQ4)

In RQ3 and RQ4, we perform a qualitative assessment of the relevance of bloated dependencies for the developers of open-source projects. We systematically select 30 notable open-source projects hosted on GitHub to conduct this analysis. We query the GitHub API to list all the Java projects ordered by their number of stars. Then, we randomly select a set of projects that fulfil all the following criteria: (1) we can build them successfully with Maven, (2) the last commit was at the latest in October 2019, (3) they declare at least one dependency in the POM, (4) they have a description in the README about how to contribute through pull requests, and (5) they have more than 100 stars on GitHub.

Table 3 shows the selected 30 projects per those criteria, to which we submitted at least one pull request. They are listed in decreasing order according to their number of stars on GitHub. The first column shows the name of the project as declared on GitHub, followed by the name of the targeted module if the project is multi-module. Notice that in the case of `jenkins` we submitted two pull requests targeting two distinct modules: `core` and `cli`. Columns two to four describe the projects according to its category as assigned to the corresponding released artifact in Maven Central, the number of commits in the master branch in October 2019, and the number of stars at the moment of conducting this study. Columns five to seven report about the total number of direct, inherited, and transitive dependencies included in the dependency tree of each considered project.

Table 3 Maven projects selected to conduct our qualitative study of bloated dependencies (RQ3 & RQ4)

Project	Description			Dependencies		
	Category	#Commits	#Stars	#D	#I	#T
jenkins [core]	Automation Server	29,040	14,578	51	2	87
jenkins [cli]	Automation Server	29,040	14,578	17	2	0
mybatis-3 [mybatis]	Relational Mapping	3,145	12,196	23	0	51
flink [core]	Streaming	19,789	11,260	14	10	34
checkstyle [checkstyle]	Code Analysis	8,897	8,897	18	0	36
auto [common]	Meta-programming	1,081	8,331	8	0	24
neo4j [collections]	Graph Database	66,602	7,069	8	2	21
CoreNLP	NLP	15,544	6,812	23	0	45
moshi [moshi-kotlin]	JSON Library	793	5,731	14	0	21
async-http-client [http-client]	HTTP Client	4,034	5,233	29	16	130
error-prone [core]	Defects Detection	4,015	4,915	44	0	35
alluxio [core-transport]	Database	30,544	4,442	6	14	73
javaparser [symbol-solver-logic]	Code Analysis	6,110	2,784	3	0	8
undertow [benchmarks]	Web Server	4,687	2,538	10	0	19
wc-capture [driver-openimaj]	Webcam	629	1,618	3	0	84
teavm [core]	Compiler	2,334	1,354	9	0	9
handlebars [markdown]	Templates	916	1,102	6	0	13
jooby [jooby]	Web Framework	2,462	1,083	23	0	68
tika [parsers]	Parsing library	4,650	929	81	0	67
orika [eclipse-tools]	Object Mapping	970	864	3	0	3
spoon [core]	Meta-programming	2,971	840	16	2	59
accumulo [core]	Database	10,314	763	26	1	51
couchdb-lucene	Text Search	1,121	752	25	0	112
jHiccup	Profiling	215	519	4	0	1
subzero [server]	Cryptocurrency	158	499	6	0	100
vulnerability-tool [shared]	Security	1,051	324	6	4	2
para [core]	Cloud Framework	1,270	310	47	2	112
launch4j-maven-plugin	Deployment Tool	316	194	7	0	61
jacop	CP Solver	1,158	155	7	0	9
selenese-runner-java	Interpreter	1,688	117	23	0	148
commons-configuration	Config library	3,159	100	31	0	49

We answer RQ3 according to the following protocol: 1) we run DepClean, we build the artifact with the debloated POM file, 2) if the project builds successfully, we analyze the project to propose a relevant change to the developers per the contribution guidelines, 3) we propose a change in the POM file in the form of a pull request, and 4) we discuss the pull

```
87  -       <dependency>
88  -           <groupId>org.apache.httpcomponents</groupId>
89  -           <artifactId>httpmime</artifactId>
90  -       </dependency>
```

Fig. 6 Example of commit removing the bloated-direct dependency `org.apache.httpcomponents:httpmime` in the project Undertow

request through GitHub. Figure 6 shows an excerpt of the diff of such a change in the POM file. We note that the submitted pull requests contain a small modification in a single file: the POM.

In the first step of the protocol, we use DepClean to obtain a report about the usage of dependencies. We analyze dependencies with both compile and test scope. Once a bloated-direct dependency is found, we remove it directly in the POM and proceed to build the project. If the project builds successfully after the removal (all the tests pass), we submit the pull request with the corresponding change. If after the removal of the dependency the build fails, then we consider the dependency as used dynamically and do not suggest removing it. In the case of multi-module projects, with bloated dependencies in several modules, we submitted a single pull request per module.

For each pull request, we analyze the Git history of the POM file to determine when the bloated dependency was declared or modified. Our objective is to collect information in order to understand how the dependencies of the projects change during their evolution. This allows us to prepare a more informative pull request message and to support our discussion with developers. We also report on the benefits of tackling these bloated dependencies by describing the differences between the original and the debloated packaged artifact of the project in terms of the size of the bundle and the complexity of its dependency tree, when the difference was significant. Each pull request includes an explanatory message. Figure 7 shows an example of the pull request message submitted to the project Undertow.[5] The message explains the motivations of the proposed change, as well as the negative impact of keeping these bloated dependencies in the project.

To answer RQ4, we follow the same pull request submission protocol as for RQ3. We use DepClean to detect bloated-transitive dependencies and submit pull requests suggesting the addition of the corresponding exclusion clauses in each project POM. Figure 8 shows an example of a pull request message submitted to the project Apache Accumulo[6], while Fig. 9 shows an excerpt of the commit proposing the exclusion of the transitive dependency `org.apache.httpcomponents:httpcore` from the direct dependency `org.apache.thrift:libthrift` in its POM.

Additional information related to the selected projects and the research methodology employed is publicly available as part of our replication package at https://github.com/castor-software/depclean-experiments.

[5] https://github.com/undertow-io/undertow

[6] https://github.com/apache/accumulo

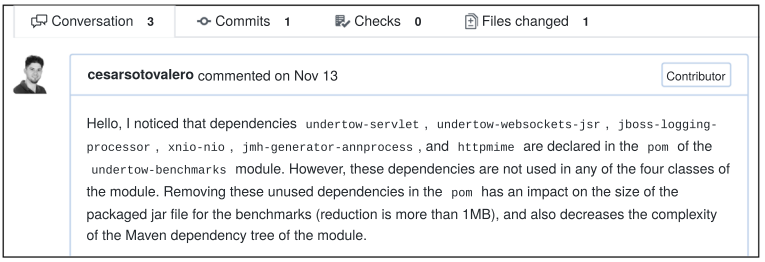


Fig. 7 Example of message of a pull request sent to the project Undertow on GitHub

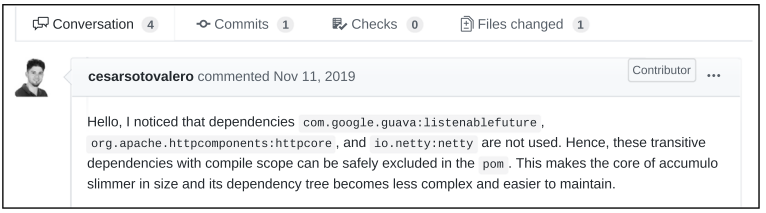


Fig. 8 Example of message in a pull request sent to the project Apache Accumulo on GitHub

```
104        <dependency>
105          <groupId>org.apache.thrift</groupId>
106          <artifactId>libthrift</artifactId>
107  +       <exclusions>
108  +         <exclusion>
109  +           <groupId>org.apache.httpcomponents</groupId>
110  +           <artifactId>httpcore</artifactId>
111  +         </exclusion>
112  +       </exclusions>
```

Fig. 9 Example of commit excluding the bloated-transitive dependency `org.apache.httpcomponents:httpcore` in the project Apache Accumulo

5 Experimental Results

We now present the results of our in-depth analysis of bloated dependencies in the Maven ecosystem.

5.1 RQ1: How Frequently do Bloated Dependencies Occur?

In this first research question, we investigate the status of all the dependency relationships of the 9,639 Maven artifacts under study.

Figure 10 shows the overall status of the 723,444 dependency relationships in our dataset. The x-axis represents the percentages, per usage type, of all the dependencies considered in the studied artifacts. The first observation is that the bloat phenomenon is massive: 543,610 (75.1%) of all dependencies are bloated, they are not needed to compile and run the code. This bloat is divided into three separate categories: 19,673 (2.7%) are bloated-direct dependency relationships (explicitly declared in the POMs); 111,649 (15.4%) are bloated-inherited dependency relationships from parent module(s); and 412,288 (57%) are bloated-transitive dependencies. Figure 10 shows that 75.1% of the relationships (edges in the dependency usage tree) are bloated dependencies. Note that this observation does not mean that 543,610 artifacts are unnecessary and can be removed from Maven Central. The same artifact can be present in several DUTs, i.e., reused by different artifacts, but be part of a bloated dependency relationship only in some of these DUTs, and part of a used relationship in the other DUTs.

Figure 11 shows the overall status of the dependencies with respect to the type of the dependency relationship (direct, inherited, and transitive). We observe that approximately 1/3 of direct dependencies are bloated (34.23%), whereas inherited and transitive dependencies have a higher percentage of bloat (61.79% and 82.5% of bloat, respectively). These results indicate that artifacts with inherited and transitive dependencies are more likely to have more bloated dependencies. They also confirm that transitive dependencies are the most susceptible to bloat.

In the following, we illustrate the three types of bloated dependency relationships with concrete examples.

Bloated-Direct We found that 2.7% of the dependencies declared in the POM file of the studied artifacts are not used at all via bytecode calls. Recall that detecting this type of bloated dependencies is good, because they are easy to remove

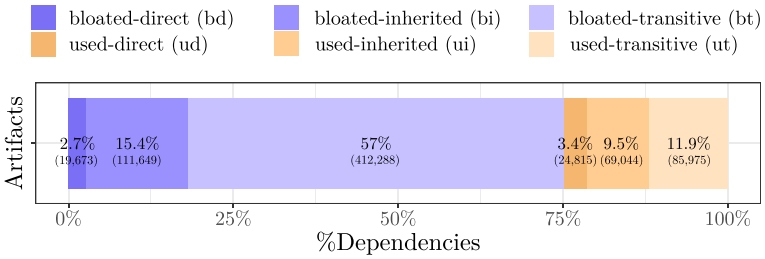


Fig. 10 Ratio per usage status of the 723,444 dependency relationships analyzed. Raw counts are inside parentheses below each percentage

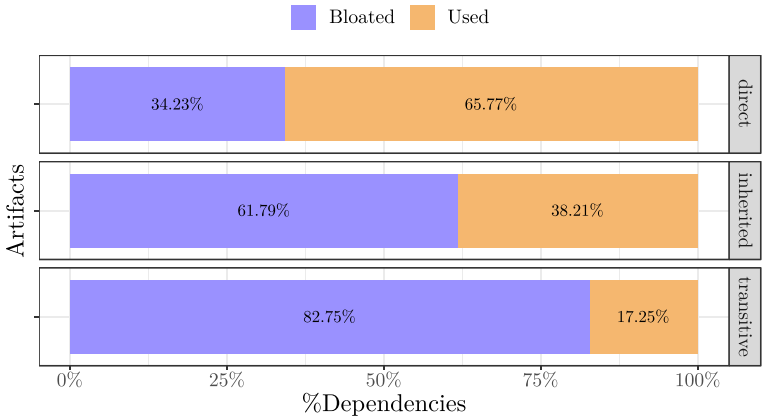


Fig. 11 Ratio per dependency type of bloated and used dependencies of the 723,444 dependency relationships analyzed

by developers with a single change in the POM file of the project under consideration. As an example, the Apache Ignite[7] project has deployed an artifact: `org.apache.ignite:ignite-zookeeper:2.4.0`, which contains only one class in its bytecode: `TcpDiscoveryZookeeperIpFinder`, and it declares a direct dependency in the POM towards `slf4j`, a widely used Java logging library. However, if we analyze the bytecode of `ignite-zookeeper`, no call to any API member of `sl4j` exists, and therefore, it is a bloated-direct dependency. After a manual inspection of the commit history of the POM, we found that `sl4j` was extensively used across all the modules of Apache Ignite at the early stages of the project, but it was later replaced by a dedicated logger, and its declaration remained intact in the POM.

Bloated-Inherited In our dataset, a total of 4,963 artifacts are part of multi-module Maven projects. Each of these artifacts declares a set of dependencies in its POM file, and also inherits a set of dependencies from a parent POM. DepClean marks those inherited dependencies are either bloated-inherited or used-inherited. Our dataset includes a total of 111,649 dependency relationships labeled as bloated-inherited, which represents 15.4% of all dependencies under study and 61.8% of the total of inherited dependencies. For example, the artifact `org.apache.drill:drill-protocol:1.14.0` inherits dependencies `commons-codec` and `commons-io` from its parent POM `org.apache.drill:drill-root:1.14.0`, however, those dependencies are not used in this module, and therefore they are bloated-inherited dependencies.

Bloated-Transitive In our dataset, bloated-transitive dependencies represent the majority of the bloated dependency relationships: 412,288 (57%). This type of bloat is a natural consequence of the Maven dependency resolution mechanism, which automatically resolves all the dependencies whether they are explicitly declared in the POM file of the project or not. Transitive dependencies are the most common type of dependency relationships, having a direct impact on the growth of the dependency trees. This type of bloat is the most common in the Maven ecosystem. For example, the artifact

[7] https://github.com/apache/ignite

```
package org.apache.streams.filters;
import com.google.common.base.Preconditions;
public class VerbDefinitionDropFilter implements
    StreamsProcessor {
  ...
  @Override
  public List<StreamsDatum> process(StreamsDatum entry) {
    ...
    Preconditions.checkArgument(entry.getDocument() instanceof
        Activity);
    return result;
  }
  ...
}
```

Listing 2 Code snippet of the class `VerbDefinitionDropFilter` present in the artifact `org.apache.streams:streams-filters:0.6.0`. The library `com.google.guava:guava:20.0` is included in its classpath via transitive dependency and called from the source code, but no dependency towards `guava` is declared in its POM

`org.eclipse.milo:sdk-client:0.2.1` ships the transitive dependency `gson` in its MDT, induced from its direct dependency towards `bsd-parser-core`. However, the part of `bsd-parser-core` used by `sdk-client` does not call any API member of `gson`, and therefore it is a bloated-transitive dependency.

In the following, we discuss the dependencies that are actually used. We observe that direct dependencies represent only 3.4% of the total of dependencies in our dataset. This means that the majority of the dependencies that are necessary to build Maven artifacts are not declared explicitly in the POM files of these artifacts.

It is interesting to note that 85,975 of the dependencies used by the artifacts under study are transitive dependencies. This kind of dependency usage occurs in two different scenarios: (1) the artifact uses API members of some transitive dependencies, without declaring them in its own POM file; or (2) the transitive dependency is necessary to provide a functionality to another, actually used dependency, in the dependency tree of the artifact.

```
package org.duracloud.audit.task;
import org.duracloud.common.json.JaxbJsonSerializer;
public class AuditTask extends TypedTask {
  ...
  private static JaxbJsonSerializer<Map<String, String>>
      getPropsSerializer() {
    return new JaxbJsonSerializer<>((Class<Map<String,
        String>>) (Object) new HashMap<String,
        String>().getClass());
  }
  ...
}
```

Listing 3 Code snippet of the class `AuditTask` present in the artifact `org.duracloud:auditor:4.4.3`. The library `org.codehaus.jackson:jackson-mapper-asl:1.6.2` is used indirectly through the direct dependency `org.duracloud:common-json:4.4.3`

We now discuss an example of the first scenario based on the `org.apache.streams:streams-filters:0.6.0` artifact from the Apache

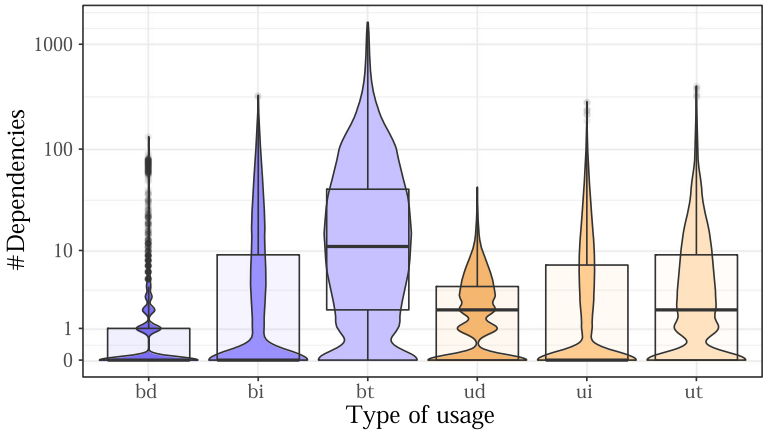


Fig. 12 Distributions of the six types of dependency usage relationships for the studied artifacts. The thicker areas on each curve represent concentrations of artifacts per type of usage

Streams[8] project. It contains two classes: `VerbDefinitionDropFilter` and `VerbDefinitionKeepFilter`. Listing 2 shows part of the source code of the class `VerbDefinitionDropFilter`, which imports the class `PreCondition` from library `guava` (line 2) and uses its static method `checkArgument` in line 8 of method `process`. However, if we inspect the POM of `streams-filters`, we notice that there is no dependency declaration towards `guava`. It declares a dependency towards `streams-core`, which in turn depends on the `streams-utils` artifact that has a direct dependency towards `guava`. Hence, `guava` is a used-transitive dependency of `streams-filters`, called from its source code.

Let us now present an example of the second scenario. Listing 3 shows an excerpt of the class `AuditTask` included in the artifact `org.duracloud:auditor:4.4.3`, from the project DuraCloud.[9] In line 6, the method `getPropsSerializer` instantiates the `JaxbJsonSerializer` object that belongs to the direct dependency `org.duracloud:common-json:4.4.3`. This object, in turn, creates an `ObjectMapper` from the transitive dependency `jackson-mapper-asl`. Hence, `jackson-mapper-asl` is a necessary, transitive provider for `org.duracloud:auditor:4.4.3`.

Figure 12 shows the distributions of dependency usage types per artifact. The figure presents superimposed log-scaled box-plots and violin-plots of the number of dependency relationships corresponding to the six usage types studied. Box-plots indicate the standard statistics of the distribution (i.e., lower/upper inter-quartile range, max/min values, and outliers), while violin plots indicate the entire distribution of the data.

We observe that the distributions of the bloated-direct (bd) and bloated-transitive (bt) dependencies vary greatly. Bloated-direct dependencies are distributed between 0 and 1 (1st-Q and 3rd-Q), with a median of 0; whereas the second ranges between 2 and 41 (1st-Q and 3rd-Q), with a median of 11. These values are in line with the statistics presented in

[8] https://streams.apache.org
[9] https://duraspace.org

Table 2, since the number of direct and transitive dependencies in general differ approximately by one order of magnitude. Overall, from the 9,639 Maven artifacts studied, 3,472 (36%) have at least one bloated-direct dependency, while 8,305 (86.2%) have at least one bloated-transitive.

On the other hand, the inter-quartile range of bloated-direct (bd) dependencies is more compact than the used-direct (ud). In other words, the dependencies declared in the POM are mostly used. This result is expected, since developers have more control over the edition (adding/removing dependencies) of the POM file of their artifact.

The median number of used-transitive (ut) dependencies is significantly lower than the median number of bloated-transitive (bt) dependencies (2, vs. 11). This suggests that the default dependency resolution mechanism of Maven is suboptimal with respect to ensuring minimal dependency inclusion.

The number of outliers in the box-plots differs for each usage type. Notably, the bloated-direct dependencies have more outliers (in total, 25 artifacts have more than 100 bloated-direct dependencies). In particular, the artifact `com.bbossgroups.pdp:pdp-system:5.0.3.9` has the maximum number of bloated-direct dependencies: 133, out of the 147 declared in its POM. The total number of artifacts with at least one bloated-direct dependency in our dataset is 2,298, which represents 23.8% of the 9,639 studied artifacts.

Answer to RQ1: The analysis of the 723,444 analysed dependency relationships in our dataset reveals that 543, 610 (75%) of them are bloated. Most of the bloated dependencies are transitive 412; 288 (57%). Overall, 36% of the artifacts have at least one bloated dependency that is declared in their POM file. To our knowledge, this is the first scientific observation of this phenomenon.
Implications: Since developers have more control over direct dependencies, up to 17, 673 (2:7%) of dependencies can be removed directly from the POM of Maven artifacts, in order to obtain smaller binaries and reduced attack surface. RQ3 will explore the willingness of developers to do so.

5.2 RQ2: How do the Reuse Practices Affect Bloated Dependencies?

In this research question, we investigate how the reuse practices that lead to these distinct types of dependency relationships are related to the bloated dependencies that emerge in Maven artifacts.

Figure 13 shows the distributions, in percentages, of the direct, inherited, and transitive dependencies for the 9,639 studied artifacts. The artifacts are sorted, from left to right, in increasing order according to their ratio of direct dependencies. The y-axis indicates the ratio of each type of dependency for a given artifact. First, we observe that 4,967 artifacts belong to multi-module projects. Among these artifacts, the extreme case (far left of the plot) is `org.janusgraph:janusgraph-berkeleyje:0.4.0`, which declares only 1.4% of its dependencies in its POM, while the 48.6% of its dependencies are inherited from parent POM files, and 50% are transitive. Second, we observe that the ratio of transitive dependencies is not equally distributed. On the right side of the plot, 879 (9.1%) artifacts have no transitive dependency (they have 100% direct dependencies). Meanwhile, 5,561 (57.7%) artifacts have more than 50% transitive dependencies. The extreme

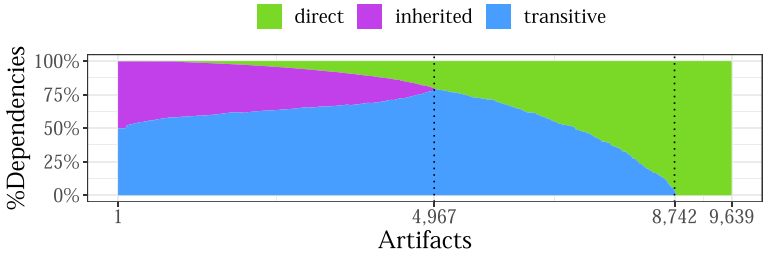


Fig. 13 Distribution of the percentages of direct, inherited, and transitive dependencies for the 9,639 artifacts considered in this study

case is `org.apereo.cas:cas-server-core-api-validation:6.1.0`, with 77.6% transitive dependencies.

In summary, the plot in Fig. 13 offers a big picture of the distribution of the three types of dependency usage in our dataset. The inherited and transitive dependencies are a significant phenomenon in Maven: 8,742 (90.7%) artifacts in our dataset have transitive dependencies, and 51.5% of artifacts belong to multi-module projects. This observation confirms the results of the previous section, most of the bloated dependencies in our dataset are either transitive (57%) or inherited (15.4%).

5.2.1 Transitive Dependencies

Figure 14 plots the relation between the ratio of transitive dependencies and the ratio of bloated dependencies. Each dot represents an artifact. Dots have a higher opacity in the presence of overlaps.

The key insight in Fig. 14 is that the larger concentration of artifacts is skewed to the top right corner, indicating that artifacts with a high percentage of transitive dependencies also tend to exhibit higher percentages of bloated dependencies. Indeed, both variables are positively correlated, according to the Spearman's rank correlation test ($\rho = 0.65$, p-value < 0.01).

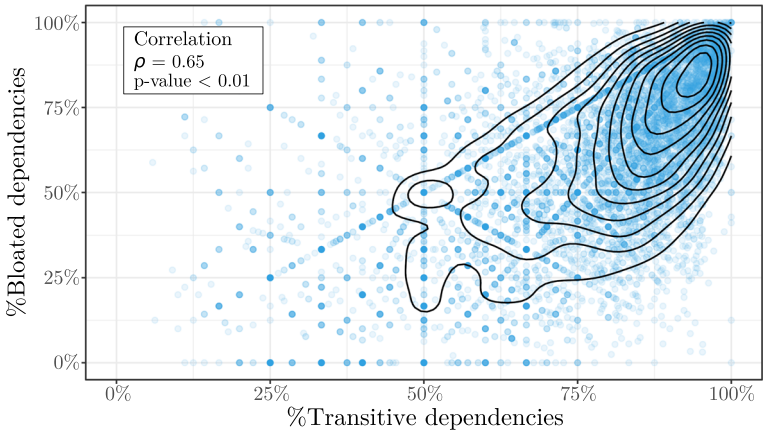


Fig. 14 Relation between the percentages of transitive dependencies and the percentage of bloated dependencies in the 9,639 studied artifacts

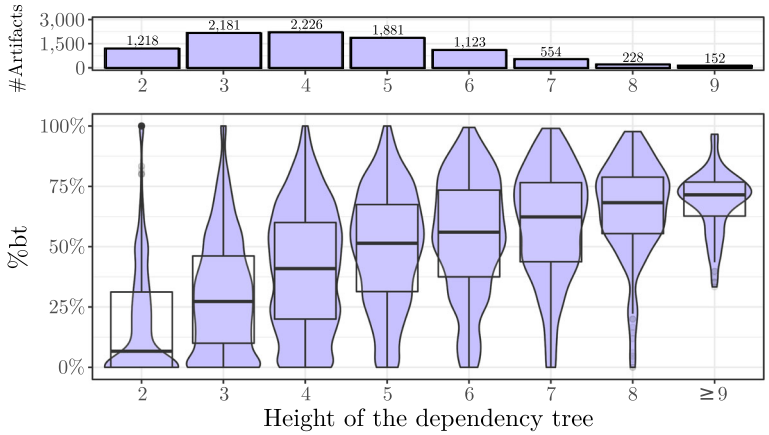


Fig. 15 Distribution of the percentages of bloated-transitive dependencies for our study subjects with respect to the height of the dependency trees. Height values greater than 10 are aggregated. The bar plot at the top represents the number of study subjects for each height

Figure 15 shows the distribution of the ratio of transitive bloated dependencies according to the height of the dependency tree. The artifact in our dataset with the largest height is `top.wboost:common-base-spring-boot-starter:3.0.RELEASE`, with a height of 14. The bar plot on top of Fig. 15 indicates the number of artifacts that have the same height. We observe that most of the artifacts have a height of 4: 2,226 artifacts in total. Considering the number of dependencies, this suggests that the dependency trees tend to be wider than deep. This is direct consequence of the automatic dependency management by Maven: any dependency that already appears at a level closer to the root will be omitted by Maven if it is referred to at a deeper level.

Looking at the 58 artifacts with height $\geq$ 9, we notice that most of them belong to multi-module projects, and declare other modules in the same project as their direct dependencies. This is a regular practice of multi-module projects, which allows to release each module as an independent artifact. Meanwhile, this increases the complexity of dependency trees. For example, artifact `org.wso2.carbon.devicemgt:org.wso2.carbon.apimgt.handlers:3.0.192` is the extreme case of this practice in our dataset, with two direct dependencies towards other modules of the same project that in turn depend on other modules of this project. As a result, this artifact has 342 bloated-transitive and 87 bloated-inherited dependencies, a dependency tree of height 11, and is part of a multi-module project with a total of 79 modules released in Maven Central.

The plot in Fig. 15 shows a clear increasing trend of bloated-transitive dependencies as the height of the dependency tree increases. Indeed, both variables are positively correlated, according to the Spearman's rank correlation test (ρ = 0.54, p-value ¡ 0.01). For artifacts with a dependency tree of height greater than 9, at least 28% of their transitive dependencies are bloated, while the median of the percentages of bloated-transitive dependencies for artifacts with height larger than 5 is more than 50%.

This finding confirms and complements the results of Fig. 14, showing that the height of the dependency tree is directly related to the occurrence of bloat. However, the height of the tree may not be the only factor that causes the bloat. For example, we hypothesize that number of transitive dependencies is another essential factor.

In order to validate this hypothesis, we perform a Spearman's rank correlation test between the number of bloated-transitive dependencies and the size of the dependency tree, i.e., the number of nodes in each tree. We found that there is a significant positive correlation between both variables (ρ =0.67, p-value < 0.01). This confirms that the actual usage of transitive dependencies decreases with the increasing complexity of the dependency tree. This result is aligned with our previous study that suggest that most of the public API members of transitive dependencies are not used by its clients (Harrand et al. 2019).

In summary, our results point to the excess of transitive dependencies as one of the fundamental causes of the existence of bloated dependencies in the Maven ecosystem.

5.2.2 Single-Module vs. Multi-Module

Let us investigate on the differences between single and multi-module architectures with respect to the presence of bloated dependencies. Figure 16 compares the distributions of bloated and used dependencies between multi-module and single-module artifacts in our dataset. We notice that, in general, multi-module artifacts have slightly more bloat than single-module, precisely 3.1% more (the percentage of bloat in single-module is $5.8\% + 67.3\% = 73.1\%$ vs. $0.9\% + 24.2\% + 51.1\% = 76.2\%$ in multi-module). More interestingly, we observe that a majority of the inherited dependencies are bloated: 24.2% of the dependencies among multi-module project are bloated-inherited (bi), while only 15% are used-inherited (ui). This suggests that most of the dependencies inherited by Maven artifacts that belong to multi-module artifacts are not used by these modules.

We observe that the percentage of bloated-direct dependencies in multi-module artifacts is very small (0.9%) in comparison with single-module (5.8%). Meanwhile, the percentage of bloated-transitive dependencies in single-module (67.3%) is larger than in multi-module (51.1%). This is due to the "shift" of a part of direct and transitive dependencies into inherited dependencies when using a parent POM. Indeed, the "shift" from direct to inherited is the main motivation for having a parent POM: to have one single declaration of dependencies for many artifacts instead of letting each artifact manage their own dependencies.

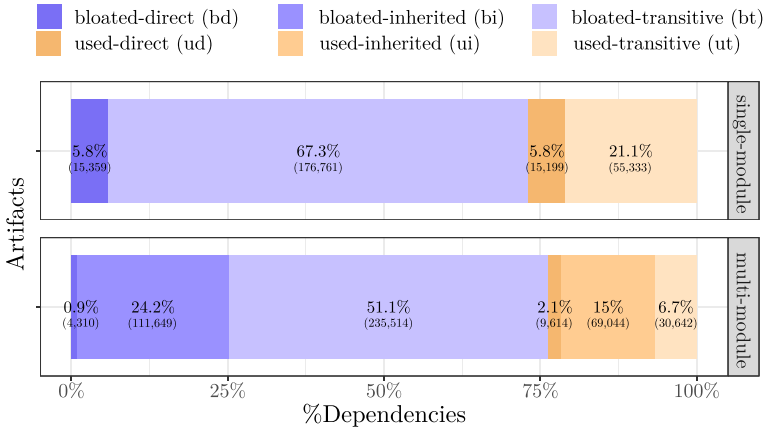


Fig. 16 Comparison between multi-module and single-module artifacts according to the percentage status of their dependency relationships. Raw counts are inside parentheses below each percentage

This "shift" in the nature of dependencies between single and multi-module artifacts is further emphasized in Fig. 17. This plot shows superimposed log scaled box-plots and violin-plots comparing the distributions of the number of distinct dependency usage types per artifact, for single-module (top part of the figure) and multi-module (bottom part).

We observe that multi-module artifacts have less bloated-direct (1st-Q = 0, median = 0, 3rd-Q = 0) and less bloated-transitive (1st-Q = 2, median = 9, 3rd-Q = 40), compared to single-modules, as shown in Fig. 17. However, multi-module artifacts have a considerably larger number of bloated-inherited dependencies instead (1st-Q = 1, median = 5, 3rd-Q = 20). The extreme case in our dataset is the artifact `co.cask.cdap:cdap-standalone:4.3.4`, with 326 bloated-inherited dependencies in total.

In summary, the multi-module architecture in Maven projects contributes to limit redundant dependencies and facilitates the consistent versioning of dependencies in large projects. However, it introduces two challenges for developers. First, it leads to the emergence of bloated-inherited dependencies because of the friction of maintaining a common parent POM file: it is more difficult to remove dependencies from a parent POM than from an artifact's own POM. Second, it is more difficult for developers to be aware of and understand the dependencies that are inherited from the parent POM. This calls for better tooling and user interfaces to help developer grasp and analyze the inherited dependencies in multi-module projects, in order to detect bloated dependencies. To our knowledge, this type of tools is absent in the current Java dependency management ecosystem.

Answer to RQ2: Reuse practices that lead to complex dependency trees and multi-module architecture are correlated with the presence of bloated dependencies: the higher a dependency tree, the more bloated-transitive dependencies. In multi-module artifacts, part of the bloat associated with direct and transitive dependencies is shifted as bloated-inherited dependencies.
Implications: Developers should carefully consider reusing artifacts with several dependencies because they introduce bloat. They should also contemplate the risks of having bloated dependencies when considering adopting a multi-module architecture.

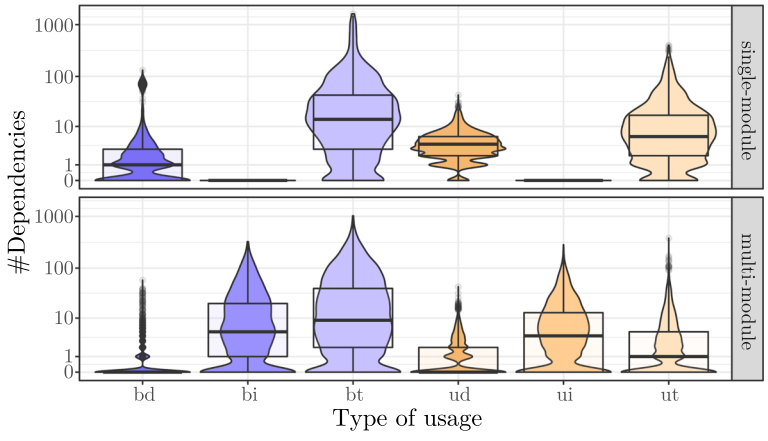


Fig. 17 Comparison between multi-module and single-module projects according to their distributions of dependency usage relationships

5.3 RQ3: To what Extent are Developers Willing to Remove Bloated-Direct Dependencies?

In this research question, our goal is to see how developers react when made aware of bloated-direct dependencies in their projects. We do this by proposing the removal of bloated-direct dependencies to lead developers of mature open-source projects, as described in Section 4.2.2.

Table 4 shows the list of 19 pull requests submitted. Each pull request proposes the removal of at least one bloated-direct dependency in the POM. We received response from developers for 17 pull request. The first and second columns in the table show the name of the project and the pull request on GitHub. Columns three and four represent the number of bloated dependencies removed in the POM and the total number of dependencies removed from the dependency tree with the proposed change, including transitive ones. The last column shows the status of the pull request (✓ accepted, ✓* accepted with changes, ✗ rejected, or ✳ pending). The last row represent the acceptance rate calculated with respect to the projects with response, i.e., the total number of dependencies removed divided by the number of proposed removals. For example, for project `undertow` we proposed the removal of 6 bloated dependencies in its module `benchmarks`. As a result of this change, 17 transitive dependencies were removed from the dependency tree the module.

Overall, from the pull requests with responses from developers, 16/17 were accepted and merged. In total, 75 dependencies were removed from the dependency trees of the projects. This result demonstrates the relevance of handling bloated-direct dependencies for developers, and the practical usefulness of DepClean.

Let us now summarize the developer feedback. First, all developers agreed on the importance of refining the projects' POMs. This is reflected in the positive comments received. Second, their quick responses suggest that it is easy for them to understand the issues associated with the presence of bloated-direct dependencies in their projects. In 8/17 projects, the response time was less than 24 hours, which is an evidence that developers consider this type of improvement as a priority.

Our results also provide evidence of the fact that we, as external contributors to those projects, were able to identify the problem and propose a solution using DepClean. In the following, we discuss four cases of pull requests that are particularly interesting and the feedback provided by developers.

5.3.1 Jenkins

DepClean detects that `jtidy` and `commons-codec` are bloated-direct dependencies present in the modules `core` and `cli` of `jenkins`. `jtidy` is an HTML syntax checker and pretty printer. `commons-codec` is an Apache library that provides an API to encode/decode from various formats such as Base64 and Hexadecimal.

Developers were reluctant to remove `jtidy` due to their concerns of affecting the users of `jenkins`, which could be potential consumers of this dependency. After further inspection, they found that the class `HTMLParser` of the `nis-notification-lamp-plugin`[10] project relies on `jtidy` transitively for performing HTML parsing.

[10] https://github.com/jenkinsci/nis-notification-lamp-plugin

Table 4 List of pull requests proposing the removal of bloated-direct dependencies created for our experiments

Project	Pull-request URL (https://github.com/)	Removed dependencies		PR*
		#D	Total	
`jenkins [core, cli]`	jenkinsci/jenkins/pull/4378	2	2	✓*
`mybatis-3 [mybatis]`	mybatis/mybatis-3/pull/1735	2	4	✓
`flink [core]`	apache/flink/pull/10386	1	1	✓
`checkstyle`	checkstyle/checkstyle/issues/7307	1	4	×
`neo4j [collections]`	neo4j/neo4j/pull/12339	1	2	✓
`CoreNLP`	stanfordnlp/CoreNLP/pull/965	1	1	✓
`async-http-client [http-client]`	AsyncHttpClient/async-http-client/pull/1675	1	12	✓
`error-prone [core]`	google/error-prone/pull/1409	1	1	✓
`alluxio [core-transport]`	Alluxio/alluxio/pull/10567	1	11	✓
`javaparser [symbol-solver-logic]`	javaparser/javaparser/pull/2403	2	9	✓
`undertow [benchmarks]`	undertow-io/undertow/pull/824	6	17	✓
`handlebars [markdown]`	jknack/handlebars.java/pull/719	1	1	✳
`jooby`	jooby-project/jooby/pull/1412	1	1	✓
`couchdb-lucene`	rnewson/couchdb-lucene/pull/279	3	3	✳
`jHiccup`	giltene/jHiccup/pull/42	1	1	✓
`subzero [server]`	square/subzero/pull/122	1	4	✓
`vulnerability-tool [shared]`	SAP/vulnerability-assessment-tool/pull/290	1	1	✓
`launch4j-maven-plugin`	lukaszlenart/launch4j-maven-plugin/pull/113	2	4	✓
`jacop`	radsz/jacop/pull/35	2	4	✓
Acceptance rate	–	25/26	75/79	16/17

Status of the pull request: ✓ Accepted. ✓ Accepted with changes. × Rejected. ✳ Pending

Developers also pointed out the fact that there is no classloader isolation in `jenkins`, and hence all dependencies in its `core` module automatically become part of its public API. A developer also referred to issues related to past experiences removing unused dependencies. He argued that external projects have depended on that inclusion and their builds were broken by such a removal. For example, the Git client plugin of `jenkins` mistakenly included Java classes from certain Apache authentication library. When they removed the dependency, some downstream consumers of the library were affected, and they had to include the dependency directly.

Consequently, we received the following feedback from an experienced developer of `jenkins`:

> We're not precluded from removing an unused dependency, but I think that the project values compatibility more than removal of unused dependencies, so we need to be careful that removal of an unused dependency does not cause a more severe problem than it solves.

After some discussions, developers agreed with the removal of `commons-codec` in module `cli`. Our pull request was edited by the developers and merged to the master branch one month after.

5.3.2 Checkstyle

DepClean identifies the direct dependency `junit-jupiter-engine` as bloated. This is a test scope dependency that was added to the POM of `checkstyle` when migrating integration tests to JUnit 5. The inclusion of this dependency was necessary due to the deprecation of `junit-platform-surefire-provider` in the Surefire Maven plugin. However, the report of DepClean about this bloated-direct dependency was a false positive. The reason for this output occurs because `junit-jupiter-engine` is commonly used through reflective calls that cannot be captured at the bytecode level.

Althoughthis pull request was rejected, developers expressed interest in DepClean, which is encouraging. They also proposed a list of features for the improvement of our tool. For example, the addition of an exclusion list in the configuration of DepClean for dependencies that are known to be used dynamically, improvements on the readability of the generated report, and the possibility of causing the build process to fail in case of detecting the presence of any bloated dependency. We implemented each of the requested functionalities in DepClean. As a result, developers opened an issue to integrate DepClean in the Continuous Integration (CI) pipeline of `checkstyle`, in order to automatically manage their bloated dependencies.[11]

5.3.3 Alluxio

DepClean detects that the direct dependency `grpc-netty`, declared in the module `alluxio-core-transport` is bloated. Figure 18 shows that this dependency also induces a total of 10 transitive dependencies that are not used (4 of them are omitted by Maven due to their duplication in the dependency tree). Developers accepted our pull request and also manifested their interest on using DepClean for managing unused dependencies in the future.

[11] https://github.com/checkstyle/checkstyle/issues/7307

```
+- io.grpc:grpc-netty:jar:1.17.1:compile
|  +- (io.grpc:grpc-core:jar:1.17.1:compile - omitted for duplicate)
|  +- io.netty:netty-codec-http2:jar:4.1.30.Final:compile
|  |  +- io.netty:netty-codec-http:jar:4.1.30.Final:compile
|  |  |  \- io.netty:netty-codec:jar:4.1.30.Final:compile
|  |  |     \- (io.netty:netty-transport:jar:4.1.30.Final:compile - omitted for duplicate)
|  |  \- io.netty:netty-handler:jar:4.1.30.Final:compile
|  |     +- io.netty:netty-buffer:jar:4.1.30.Final:compile
|  |     |  \- io.netty:netty-common:jar:4.1.30.Final:compile
|  |     +- (io.netty:netty-transport:jar:4.1.30.Final:compile - omitted for duplicate)
|  |     \- (io.netty:netty-codec:jar:4.1.30.Final:compile - omitted for duplicate)
```

Fig. 18 Transitive dependencies induced by the bloated-direct dependency `grpc-netty` in the dependency tree of module `alluxio-core-transport`. The tree is obtained with the `dependency:tree` Maven goal

5.3.4 Undertow

DepClean detects a total of 6 bloated-direct dependencies in the `benchmarks` module of the project `undertow`: `undertow-servlet`, `undertow-websockets-jsr`, `jboss-logging-processor`, `xnio-nio`, `jmh-generator-annprocess`, and `httpmime`. In this case, we received a rapid positive response from the developers two days after the submission of the pull request. Removing the suggested bloated-direct dependencies has a significant impact on the size of the packaged JAR artifact of the `undertow-benchmarks` module. We compare the sizes of the bundled JAR before and after the removal of those dependencies: the binary size reduction represents more than 1MB. It is worth mentioning that this change also reduced the complexity of the dependency tree of the module.

Summary of RQ3: We used DepClean to propose 19 pull requests removing bloated-direct dependencies, from which 17/19 were answered. 16/17 pull requests with response were accepted and merged by open-source developers (In total, 75 dependencies were removed from the dependency tree of 16 projects).
Implications: Removing bloated-direct dependencies is relevant for developers and it is perceived as a valuable contribution. This type of change in the POM files are small, and they can have a significant impact on the dependency tree of Maven projects.

5.4 RQ4: To what Extent are Developers Willing to Exclude Bloated-Transitive Dependencies?

In this research question, our goal is to see how developers react when made aware of bloated-transitive dependencies. We do this by proposing the exclusion of bloated-transitive dependencies to them, as described in Section 4.2.2.

Table 5 shows the list of 13 pull requests submitted. Each pull request proposes the exclusion of at least one transitive dependency in the POM. We received response from developers for 9 pull requests. The first and second columns show the name of the project and the pull request on GitHub. Columns three and four represent the number of bloated-transitive dependencies explicitly excluded and the total number of dependencies removed in the dependency tree as resulting from the exclusion. The last column shows the status of the pull request (✓ accepted, × rejected, or ✳ pending). The last row represents the acceptance rate with respect to the projects with response. For example, for the project `spoon` we propose the exclusion of four bloated-transitive dependencies in its `core` module. As a

result of this change, 31 transitive dependencies were removed from the dependency tree of this module.

Overall, from the pull requests with responses from developers, 5 were accepted and 4 were rejected. In total, 65 bloated dependencies were removed from the dependency trees of 5 projects. We notice that the accepted pull requests involve those projects for which the exclusion of transitive dependencies also represents the removal of a large number of other dependencies from the dependency tree. This result suggests that developers are more careful concerning this type of contribution.

As in RQ3, we obtained valuable feedback from developers about the pros and cons of excluding bloated-transitive dependencies. In the following, we provide unique qualitative insights about the most interesting cases and explain the feedback obtained from developers to the research community.

5.4.1 Jenkins

DepClean detects the bloated-transitive dependencies `constant-pool-scanner` and `eddsa` in the module `core` of `jenkins`. These bloated dependencies were induced through the direct dependencies `remoting` and `cli`, respectively. In the message of the pull request, we explain how their exclusion contributes to make the `core` of `jenkins` slimmer and its dependency tree clearer.

Although both dependencies were confirmed as unused in the `core` module of `jenkins`, developers rejected our pull request. They argue that excluding such dependencies has no valuable repercussion for the project and might actually affect its clients, which is correct. For example, `constant-pool-scanner` is used by external components, e.g., the class `RemoteClassLoader` in the `remoting`[12] project relies on this library to inspect the bytecode of remote dependencies.

As shown in the following quote from an experienced developer of Jenkins, there is a consensus on the usefulness of removing bloated dependencies, but developers need strong facts to support the removal of transitive dependencies:

> Dependency removals and exclusions are really useful, but my recommendation would be to avoid them if there is no substantial gain.

5.4.2 Auto

DepClean reports on the bloated-transitive dependencies `listenablefuture` and `auto-value-annotations` in module `auto-common` of the Google `auto` project. We proposed the exclusion of these dependencies and submitted a pull request with the POM change.

Developers express several concerns related to the exclusion of these dependencies. For example, a developer believes that it is not worth maintaining exclusion lists for dependencies that cause no problem. They point out that although `listenableFuture` is a single class file dependency, its presence in the dependency tree is vital to the project, since it overrides the version of the `guava` library that have many classes. Therefore, the inclusion of this dependency is a strategy followed by `guava` to narrow the access to the interface `ListenableFuture` and not to the whole library.[13]

[12] https://github.com/jenkinsci/remoting

[13] https://groups.google.com/forum/#!topic/guava-announce/Km82fZG68Sw/discussion

Table 5 List of pull requests proposing the exclusion of bloated-transitive dependencies

Project	Pull-request URL (https://github.com/)	Excluded dependencies		PR*
		#T	Total	
`jenkins [core]`	jenkinsci/jenkins/pull/4378	2	2	✗
`auto [common]`	google/auto/pull/789	2	2	✗
`moshi [moshi-kotlin]`	square/moshi/pull/1034	3	3	✗
`spoon [core]`	INRIA/spoon/pull/3167	4	31	✓
`moshi [moshi-kotlin]`	square/moshi/pull/1034	3	3	✗
`wc-capture [driver-openimaj]`	sarxos/webcam-capture/pull/750	1	1	✳
`teavm [core]`	konsoletyper/teavm/pull/439	1	2	✳
`tika [parsers]`	apache/tika/pull/299	1	2	✳
`orika [eclipse-tools]`	orika-mapper/orika/pull/328	1	2	✓
`accumulo [core]`	apache/accumulo/pull/1421	3	3	✓
`para [core]`	Erudika/para/pull/69	1	20	✓
`selenese-runner-java`	vmi/selenese-runner-java/pull/313	2	9	✓
`commons-configuration`	apache/commons-configuration/pull/40	2	9	✳
Acceptance rate	–	11/27	65/75	5/9

*Status of the pull request: ✓ Accepted. ✗ Rejected. ✳ Pending

On the other hand, developers agree that `auto-value-annotations` is bloated. However, they keep it, arguing that it is a test-only dependency, and they prefer to keep annotation-only dependencies and let end users exclude them when desired.

The response from developers suggests that bloated dependencies with test scope are perceived as less harmful. This is reasonable since test dependencies are only available during the test, compilation, and execution phases and are not shipped transitively in the JAR of the artifact. However, we believe that although it is a developers' decision whether they keep this type of bloated dependency or not, the removal of testing dependencies is regularly a desirable refactoring improvement.

5.4.3 Moshi

DepClean detects that the bloated-transitive dependency `kotlin-stdlib-common` is present in the dependency tree of modules `moshi-kotlin`, `moshi-kotlin-codegen`, and `moshi-kotlin-tests` of project `moshi`. This dependency is induced from a common dependency of these modules: `kotlin-stdlib`.

Developers rejected our pull requests, arguing that excluding such transitive dependency prevents the artifacts from participating in the proper dependency resolution of their clients. They suggest that clients interested in reducing the size of their projects can use specialized shrinking tools, such as ProGuard,[14] for this purpose.

Although the argument of developers is valid, we believe that delegating the task of bloat removal to their library clients imposes an unnecessary burden on them. On the other hand, recent studies reveal that library clients do not widely adopt the usage of dependency analysis tools for quality analysis purposes (Nguyen et al. 2020).

5.4.4 Spoon

DepClean detects that the transitive dependencies `org.eclipse.core.resources`, `org.eclipse.core.runtime`, `org.eclipse.core.filesystem`, and `org.eclipse.text` `org.eclipse.jdt.core` are bloated. All of these transitive dependencies were induced by the inclusion of the direct dependency `org.eclipse.jdt.core`, declared in the POM of `core` module of the `spoon` library.

Table 6 shows how the exclusion of these bloated-transitive dependencies has a positive impact on the size and the number of classes of the library. As we can see, by excluding these dependencies the size of the `jar-with-dependencies` of the `core` module of `spoon` is trimmed from 16.2MB to 12.7MB, which represents a significant reduction in size of 27.6%. After considering this improvements, the developers confirmed the relevance of this change and merged our pull request into the master branch of the project.

5.4.5 Accumulo

DepClean detects the bloated-transitive dependencies `listenablefuture`, `httpcore` and `netty` in the `core` module of Apache `accumulo`. These dependencies were confirmed as bloated by the developers. However, they manifested their concerns regarding their exclusion, as expressed in the following comment:

[14] https://www.guardsquare.com/en/products/proguard

Table 6 Comparison of the size and number of classes in the bundled JAR of the `core` module of `spoon`, before and after the exclusion of bloated-transitive dependencies

	JAR Size(MB)	#Classes
Before	16.2	7,425
After	12.7	5,593
Reduction(%)	27.6%	24.7%

> I'm not sure I want us to take on the task of maintaining an exclusion set of transitive dependencies from all our deps POMs, because those can change over time, and we can't always know which transitive dependencies are needed by our dependencies.

After the discussion, developers decided to accept and merge the pull request. Overall, developers considered that the proposal is a good idea. They suggest that it would be better to approach the communities of each of the direct dependencies that they use, and encourage them to mark those dependencies as *optional*, thus they would not be automatically inherited by their users.

5.4.6 Para

DepClean detects the bloated-transitive dependency `flexmark-jira-converter`. This dependency is induced through the direct dependency `flexmark-ext-emoji`, declared in the `core` module of the `para` project. Our further investigation on the Maven dependency tree of this module revealed that this bloated dependency adds a total of 19 additional dependencies to the dependency tree of the project, of which 15 are detected as duplicated by Maven.

Because of this large number (19) of bloated-transitive dependencies removed, developers accepted the pull request and merged the change into the master branch of the project the same day of the pull request submission.

Answer to RQ4: We used DepClean to propose 13 pull requests to exclude bloated-transitive dependencies, with 9 answered, 5/9 pull requests with response were accepted and merged by developers (65 dependencies were removed from the dependency tree of 5 projects).
Implications: The handling of bloated-transitive dependencies is a topic with no clear consensus among developers. Developers consider this kind of bloat as relevant, but they are concerned about the maintenance of a list of exclusion directives. Some developers agree to remove them based on practical facts (e.g., JAR size reduction), while other developers prefer to keep bloated-transitive dependencies in order to avoid the potential negative impact on their clients.

6 Discussion

In this section, we discuss the implications of our findings and the threats to the validity of the results obtained.

6.1 Implications of Results

Our results indicate that most of the dependency bloat is due to transitive dependencies and the Maven dependency inheritance mechanism. This suggests that the Maven dependency resolution strategy, which always picks the dependency that is closer to the root of the tree, may not be the best selection criterion for minimizing transitive dependency bloat. The official Maven dependency management guidelines[15] encourage developers to take control over the dependency resolution process via explicit declaration of dependencies in the POM file. This is a good practice to provide better documentation for the project and to keep one's artifact dependencies independent of the choices of other libraries down the dependency tree. Dependencies declared in this way have priority over the Maven mediation mechanism, allowing developers to have a clear knowledge about which library version they are expecting to be used through transitive dependencies. However, since backward compatibility is not always guaranteed, having fixed transitive dependency versions, and therefore non-declared dependencies, still remains as a widely accepted practice. In this context, the introduction of the module construct in Java 9 provides a higher level of aggregation above packages. This new language element, if largely adopted, may help to reduce the transitive explosion of dependencies. Indeed, this mechanism enables developers to fine tune public access restrictions of API members, explicitly declaring what set of functionalities a module can expose to other modules. This leads to two benefits: (1) it enables reuse declaration at a finer grain than dependencies, and (2) it makes the debloat techniques described in this work safer as it constrains reflection to white-listed modules.

Our results show that even notable open-source projects, which are maintained by development communities with strict development rules, are affected by dependency bloat. Developers confirmed and removed most of the reported bloated-direct dependencies detected by DepClean. However, they are more careful about excluding bloated-transitive dependencies. The addition of exclusion clauses to the POM files is perceived by some developers as an unnecessary maintainability burden. Interestingly, our quantitative results indicate that bloated-transitive dependency relationships represent the largest portion of bloated dependencies, yet, our qualitative study reveals that these bloated relationships are also the ones that developers find the most challenging to handle and reason about. Overall, this work opens the door to new research opportunities on debloating POMs and other build files.

6.2 Threats to Validity

In the following, we discuss construct, internal and external threats to the validity of our study.

Construct Validity The threats to construct validity are related to the novel concept of bloated dependencies and the metrics utilized for its measurement. For example, the DUT constructed by DepClean could be incomplete due to issues during the resolution of the dependencies. We mitigate this threat by building DepClean on top of Maven plugins to collect the information about the dependency relationships. We also exclude from the study those artifacts for which we were unable to retrieve the full dependency usage information.

[15] https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

It is possible that developers repackage a library as a bundle JAR file along with its dependencies, or copy the source code of dependencies directly into their source code, in order to avoid dependency related issues. Consequently, DepClean will miss such dependencies, as they are not explicitly declared in the POM file. Thus, the analysis of dependencies can underestimate the part of bloated dependencies. However, considering the size of our dataset and the feedback obtained from actively maintained projects, we believe that these corner cases do not affect our main results.

Internal Validity The threats to internal validity are related to the effectiveness of DepClean to detect bloated dependencies. The dynamic features of the Java programming language, e.g., reflection or dynamic class loading present particular challenges for any automatic analysis of Java source code (Landman et al. 2017; Lindholm et al. 2014). Since DepClean statically analyzes bytecode, anything that does not get into the bytecode is not detected (e.g., constants, annotations with source-only retention police, links in Javadocs), which can lead to false positives. To mitigate this threat, DepClean can detect classes or class members that are created or invoked dynamically using basic constructs such as `class.forName("someClass")` or `class.getMethod("someMethod", null)`.

To evaluate the impact of this limitation in practice, we ran DepClean on 10 additional popular projects. The experiment consists in running the test suite of the projects with the debloated version of the POM files, i.e., relying on dynamic analysis as a validation mechanism. Table 7 shows the results obtained after running the test suite of the version of the project without bloated dependencies. The first column shows the URL of the project on GitHub, the second and third columns represent the number bloated-direct and bloated-transitive dependencies detected by DepClean, and the fourth column is the result of the test (✓ pass, or ✗ fail). As we observe, 9/10 projects pass the test suite, and only one project fails: `raft-java`. We found that the reason of the failure was the dependency `org.projectlombok:lombok:1.18.4`, which heavily relies on reflection and other dynamic mechanisms of Java. To prevent the occurrence of false positives, the users of DepClean can add dependencies that are known to be used only dynamically to an exclusion list. Once added this dependency to the exclusion list of DepClean, it is not considered as bloated, and all the tests pass with the other bloated dependencies removed.

Table 7 Evaluation of the results of DepClean by checking if all the test pass after the removal of bloated dependencies

URL (https://github.com/)	#bd	#bt	Test result
pf4j/pf4j	3	3	✓
apilayer/restcountries	5	13	✓
modelmapper/modelmapper	2	14	✓
xtuhcy/gecco	0	3	✓
yaphone/itchat4j	1	1	✓
electronicarts/ea-async	0	7	✓
twitter/hbc	0	1	✓
skyscreamer/JSONassert	0	1	✓
wenweihu86/raft-java	2	8	✗
liaochong/myexcel	0	2	✓

External Validity The relevance of our findings in other software ecosystems is one threat to external validity. Our observations about bloated dependencies are based on Java and the Maven ecosystem and our findings are restricted to this scope. More studies on other dependency management systems are needed to figure out whether our findings can be generalized. Another external threat relates to the representativeness of the projects considered for the qualitative study. To mitigate this threat, we submitted pull requests to a set of diverse, mature, and popular open-source Java projects that belong to distinct communities and cover various application domains. This means that we contributed to improving the dependency management of projects that are arguably among the best of the open-source Java world, which aims to get as strong external validity as possible.

7 Related Work

In this work, we propose the first systematic large-scale analysis of bloat in the Maven ecosystem. Here, we discuss the related works in the areas of software debloating and dependency management.

7.1 Analysis and Mitigation of Software Bloat

Previous studies have shown that software tends to grow over time, whether or not there is a need for it (Holzmann 2015; Quach et al. 2017). Consequently, software bloat appears as a result of the natural increase of software complexity, e.g., the addition of non-essential features to programs (Brooks 1987). This phenomenon comes with several risks: it makes software harder to understand and maintain, increases the attack surface, and degrades the overall performance. Our paper contributes to the analysis and mitigation of a novel type of software bloat: *bloated dependencies*.

Celik et al. (2016) presented MOLLY, a build system to lazily retrieve dependencies in CI environments and reduce build time. For the studied projects, the build time speed-up reaches 45% on average compared to Maven. DEPCLEAN operates differently than MOLLY: it is not an alternative to Maven as MOLLY is, but a static analysis tool that allows Maven users to have a better understanding and control about their dependencies.

Yu et al. (2003) investigated the presence of unnecessary dependencies in header files of large C projects. Their goal was to reduce build time. They proposed a graph-based algorithm to statically remove unused code from applications. Their results show a reduction of build time of 89.70% for incremental builds, and of 26.38% for fresh builds. Our work does not focus on build performance, we analyze the pervasiveness of dependency bloat across a vast and modern ecosystem of Maven packages.

In recent years, there has been a notable interest in the development of debloating techniques for program specialization. The aim is to produce a smaller, specialized version of programs that consume fewer resources while hardening security (Azad et al. 2019). They range from debloating command line programs written in C (Sharif et al. 2018), to the specialization of JavaScript frameworks (Vázquez et al. 2019) and fully fledged containerized platforms (Rastogi et al. 2017). Most debloating techniques are built upon static analysis and are conservative in the sense that they focus on trimming unreachable code (Jiang et al. 2016), others are more aggressive and utilize advanced dynamic analysis techniques to remove potentially reachable code (Heath et al. 2019). Our work addresses the same challenges at a coarser granularity. DEPCLEAN removes unused dependencies, which is, according to our empirical results, a significant cause of program bloat.

Qiu et al. (2016) empirically show evidence that a considerable proportion of API members are not widely used, i.e., many classes, methods, and fields of popular libraries are not used in practice. (Pham et al. 2016) implement a bytecode based analysis tool to learn about the actual API usage of Android frameworks. Hejderup (2015) study the actual usage of modules and dependencies in the Rust ecosystem, and propose PRÄZI, a tool for constructing fine-grained call-based dependency networks (Hejderup et al. 2018). Lämmel et al. (2011) perform a large-scale study on API usage based on the migration of AST code segments. Other studies have focused on understanding how developers use APIs on a daily basis (Roover et al. 2013; Bauer et al. 2014). Some of the motivations include improving API design (Myers and Stylos 2016; Harrand et al. 2019) and increasing developers productivity (Lim 1994). All these studies hint at the presence of bloat in APIs. To sum up, our paper is the first empirical study that explores and consolidates the concept of bloated dependencies in the Maven ecosystem, and is the first to investigate the reaction of developers to bloated dependencies.

Program slicing (Horwitz et al. 1988; Sridharan et al. 2007; Binkley et al. 2019) is a program analysis technique used to compute the subset of statements ("slice") that affect the values of a given program. Static slicing removes unused code by computing a statement-based dependence graph and identifies the statements that are directly or transitively reachable from a seed on the graph. DepClean uses a similar approach for debloat, where the slices are bytecode calls between dependencies computed by backtracking usages between the artifact and its dependencies.

7.2 Dependency Management in Software Ecosystems

Library reuse and dependency management has become mainstream in software development. McIntosh et al. (2012) analyze the evolution of automatic build systems for Java (ANT and Maven). They found that Java build systems follow linear or exponential evolution patterns in terms of size and complexity. In this context, we interpret bloated dependencies as a consequence of the tendency of build automation systems of evolving towards open-ended complexity over time.

Decan et al. (2019, 2017) studied the fragility of packaging ecosystems caused by the increasing number of transitive dependencies. Their findings corroborate our results, showing that most clients have few direct dependencies but a high number of transitive dependencies. They also found that popular libraries tend to have larger dependency trees. However, their work focuses primarily on the relation between the library users and their direct providers and does not take into account the inherited or transitive dependencies of those providers. We are the first, to the best of our knowledge, to conduct an empirical analysis of bloated dependencies in the Maven ecosystem considering both, users and providers, as potential sources of software bloat.

Bezemer et al. (2017) performed a study of unspecified dependencies, i.e., dependencies that are not explicitly declared in the build systems. They found that these unspecified dependencies are subtle and difficult to detect in make-based build systems. Seo et al. (2014) analyzed over 26 millions builds in Google to investigate the causes, types of errors made, and resolution efforts to fix the failing builds. Their results indicate that, independent of the programming language, dependency errors are the most common cause of failures, representing more than two thirds of fails for Java. Based on our results, we hypothesize that removing dependency bloat would reduce spurious CI errors related to dependencies.

Jezek and Dietrich (2014) describe, with practical examples, the issues caused by transitive dependencies in Maven. They propose a static analysis approach for finding missing, redundant, incompatible, and conflicting API members in dependencies. Their experiments, based on a dataset of 29 Maven projects, show that problems related to transitive dependency are common in practice. They identify the use of wrong dependency scopes as a primary cause of redundancy. Our quantitative study extends this work to the scale of the Maven Central ecosystem, and provides additional evidence about the persistence of the dependency redundancy problem.

Callo Arias et al. (2011) performed a systematic review about dependency analysis solutions in software-intensive systems. Bavota et al. (2015) studied performed an empirical study on the evolution of declared dependencies in the Apache community. They found that build system specifications tend to grow over time unless explicit effort is put into refactoring them. Our qualitative results complement previous studies that present empirical evidence that developers do not systematically update their dependency configuration files (McIntosh et al. 2014; Kula et al. 2018).

8 Conclusion

In this work, we presented a novel conceptual analysis of a phenomenon originated from the practice of software reuse, which we coined as *bloated dependencies*. This type of dependency relationship between software artifacts is intriguing: from the perspective of the dependency management systems that are unable to avoid it, and from the standpoint of developers who declare dependencies but do not use them in their applications.

We performed a quantitative and qualitative study of bloated dependencies in the Maven ecosystem. To do so, we implemented a tool, DepClean, which analyzes the bytecode of an artifact and all its dependencies that are resolved by Maven. As a result of the analysis, DepClean provides a report of the bloated dependencies, as well as a new version of its POM file which removes the bloat. We use DepClean to analyze the 723,444 dependency relationships of 9,639 artifacts in Maven Central. Our results reveal that 75.1% of them are bloated (2.7% are direct dependencies, 15.4% are inherited from parent POMs, and 57% are transitive dependencies). Based on these results, we distilled two possible causes: the cascade of unwanted transitive dependencies induced by direct dependencies, and the dependency heritage mechanism of multi-module Maven projects.

We complemented our quantitative study of bloated dependencies with an in-depth qualitative analysis of 30 mature Java projects. We used DepClean to analyze these projects and submitted the results obtained as pull request on GitHub. Our results indicated that developers are willing to remove bloated-direct dependencies: 16 out of 17 answered pull requests were accepted and merged by the developers in their code base. On the other hand, we found that developers tend to be skeptical regarding the exclusion of bloated-transitive dependencies: 5 out of 9 answered pull requests were accepted. Overall, the feedback from developers revealed that the removal of bloated dependencies clearly worth the additional analysis and effort.

Our study stresses the need to engineer, i.e., analyze, maintain, test POM files. The feedback from developers shows interest in DepClean to address this challenge. While the tool is robust enough to analyze a variety of real-world projects, developers now ask questions related to the methodology for dependency debloating, e.g., when to analyze bloat? (in every build, in every release, after every POM change), who is responsible for debloat of direct

or transitive dependencies? (the lead developers, any external contributor), how to properly managing complex dependency trees to avoid dependency conflicts? These methodological questions are part of the future work to further consolidate DEPCLEAN.

Acknowledgments This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

Funding Open Access funding provided by Royal Institute of Technology

References

Azad BA, Laperdrix P, Nikiforakis N (2019) Less is more: Quantifying the security benefits of debloating web applications. In: Proceedings of the 28th USENIX conference on security symposium, SEC, pp 1697–1714, USA, USENIX Association

Bauer V, Eckhardt J, Hauptmann B, Klimek M (2014) An exploratory study on reuse at Google. In: Proceedings of the 1st International workshop on software engineering research and industrial practices, SERIP. ACM, New York, pp 14–23

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015) How the apache community upgrades dependencies: An evolutionary study. Empir Softw Eng 20(5):1275–1317

Benelallam A, Harrand N, Soto-Valero C, Baudry B, Barais O (2019) The Maven dependency graph: a temporal graph-based representation of Maven Central. In: 16th international conference on mining software repositories (MSR). IEEE/ACM, Montreal

Bezemer C.-P., McIntosh S, Adams B, German DM, Hassan AE (2017) An empirical study of unspecified dependencies in make-based build systems. Empir Softw Eng 22(6):3117–3148

Binkley D, Gold N, Islam S, Krinke J, Yoo S (2019) A comparison of tree-and line-oriented observational slicing. Empir Softw Eng 24(5):3077–3113

Brooks FP (1987) No silver bullet: Essence and accidents of software engineering. Computer 20(4):10–19

Callo Arias TB, van der Spek P, Avgeriou P (2011) A practice-driven systematic review of dependency analysis solutions. Empir Softw Eng 16(5):544–586

Celik A, Knaust A, Milicevic A, Gligoric M (2016) Build system with lazy retrieval for java projects. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE,. ACM, New York, pp 643–654

Cox R (2019) Surviving software dependencies. Commun ACM 62(9):36–43

Decan A, Mens T, Claes M (2017) An empirical comparison of dependency issues in OSS packaging ecosystems. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering, SANER, pp 2–12

Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir Softw Eng 24(1):381–416

Gkortzis A, Feitosa D, Spinellis D (2019) A double-edged sword? Software reuse and potential security vulnerabilities. In: Reuse in the big data era. Springer International Publishing, pp 187–203

Harrand N, Benelallam A, Soto-Valero C, Barais O, Baudry B (2019) Analyzing 2.3 million Maven dependencies to reveal an essential core in APIs. arXiv:1908.09757

Heath B, Velingker N, Bastani O, Naik M (2019) PolyDroid: Learning-driven specialization of mobile applications. arXiv:1902.09589

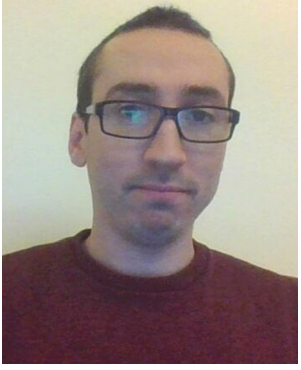
Hejderup J (2015) In dependencies we trust: How vulnerable are dependencies in software modules? PhD thesis, Delft University of Technology

Hejderup J, Beller M, Gousios G (2018) PRÄZI: From package-based to precise call-based dependency network analyses. Delft University of Technology
Holzmann GJ (2015) Code inflation. IEEE Softw 1(2):10–13
Horwitz S, Reps T, Binkley D (1988) Interprocedural slicing using dependence graphs. SIGPLAN Not 23(7):35–46
Jezek K, Dietrich J (2014) On the use of static analysis to safeguard recursive dependency resolution. In: 2014 40th EUROMICRO conference on software engineering and advanced applications, pp 166–173
Jiang Y, Wu D, Liu P (2016) JRed: Program customization and bloatware mitigation based on static analysis. In: 2016 IEEE 40th annual computer software and applications conference (COMPSAC), vol 1, pp 12–21
Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183
Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empir Softw Eng 23(1):384–417
Lämmel R, Pek E, Starek J (2011) Large-scale, AST-based API-usage analysis of open-source java projects. In: Proceedings of the 2011 ACM symposium on applied computing, SAC '11. ACM, New York, pp 1317–1324
Landman D, Serebrenik A, Vinju JJ (2017) Challenges for static analysis of java reflection - literature review and empirical study. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 507–518
Lim WC (1994) Effects of reuse on quality, productivity, and economics. IEEE Softw 11(5):23–30
Lindholm T, Yellin F, Bracha G, Buckley A (2014) The java virtual machine specification. Pearson Education
McIntosh S, Adams B, Hassan AE (2012) The evolution of java build systems. Empir Softw Eng 17(4):578–608
McIntosh S, Poehlmann M, Juergens E, Mockus A, Adams B, Hassan AE, Haupt B, Wagner C (2014) Collecting and leveraging a benchmark of build system clones to aid in quality assessments. In: Companion proceedings of the 36th international conference on software engineering, ICSE Companion 2014. Association for Computing Machinery, New York, pp 145–154
Myers BA, Stylos J (2016) Improving API usability. Commun ACM 59(6):62–69
Naur P., Randell B. (eds) (1969) Software engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968, Brussels, Scientific Affairs Division, NATO. Newcastle University, Newcastle upon Tyne
Nguyen PT, Di Rocco J, Di Ruscio D, Di Penta M (2020) CrossRec: Supporting software developers by recommending third-party libraries. J Syst Softw 161:110460
Pham HV, Vu PM, Nguyen TT et al (2016) Learning API usages from bytecode: A statistical approach. In: Proceedings of the 38th international conference on software engineering. ACM, pp 416–427
Qiu D, Li B, Leung H (2016) Understanding the API usage in Java. Info Softw Technol 73:81–100
Quach A, Erinfolami R, Demicco D, Prakash A (2017) A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments. In: Proceedings of the 2017 workshop on forming an ecosystem around software transformation. ACM, pp 65–70
Rastogi V, Davidson D, De Carli L, Jha S, McDaniel P (2017) Cimplifier: Automatically debloating containers. In: Proceedings of the 2017 11th Joint meeting on foundations of software engineering, ESEC/FSE 2017. ACM, New York, pp 476–486
Roover CD, Lämmel R, Pek E (2013) Multi-dimensional exploration of API usage. In: 21st International conference on program comprehension, ICPC, pp 152–161
Salza P, Palomba F, Di Nucci D, De Lucia A, Ferrucci F (2019) Third-party libraries in mobile apps. Empirical Software Engineering
Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R (2014) Programmers' build errors: A case study (at Google). In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 724–734
Sharif H, Abubakar M, Gehani A, Zaffar F (2018) TRIMMER: Application specialization for code debloating. In: Proceedings of the 33rd ACM/EEE international conference on automated software engineering, ASE 2018. ACM, New York, pp 329–339
Shull F, Singer J, Sjøberg DI (2007) Guide to advanced empirical software engineering. Springer, Berlin
Soto-Valero C, Benelallam A, Harrand N, Barais O, Baudry B (2019) The emergence of software diversity in Maven Central. In: 16th international conference on mining software repositories, MSR 2019. ACM, New York, pp 1–10
Sridharan M, Fink SJ, Bodik R (2007) Thin slicing. In: Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation, PLDI'07. Association for Computing Machinery, New York, pp 112–122
Vázquez H, Bergel A, Vidal S, Pace JD, Marcos C (2019) Slimming Javascript applications: An approach for removing unused functions from Javascript libraries. Inf Softw Technol 107:18–29

Wu Y, Manabe Y, Kanda T, German DM, Inoue K (2017) Analysis of license inconsistency in large collections of open source projects. Empir Softw Eng 22(3):1194–1222

Yu Y, Dayani-Fard H, Mylopoulos J (2003) Removing false code dependencies to speedup software build processes. In: Proceedings of the 2003 conference of the centre for advanced studies on collaborative research, CASCON. IBM Press, pp 343–352

César Soto-Valero is a Ph.D. student in Software Engineering at KTH Royal Institute of Technology, Sweden. His research work focuses on leveraging static and dynamic program analysis techniques to mitigate software bloat. César received his MSc degree and BSc degree in Computer Science from Universidad Central "Marta Abreu" de Las Villas, Cuba.



Nicolas Harrand is a Ph.D. student in Software Engineering at KTH Royal Institute of Technology, Sweden. His current research is focused on automatic software diversification. Nicolas studied Computer Science and Applied Mathematics in Grenoble, France.

Martin Monperrus is a Professor of Software Technology at KTH Royal Institute of Technology. He was previously an associate professor at the University of Lille and an adjunct researcher at Inria. He received a Ph.D. from the University of Rennes and a Master's degree from the Compiègne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, program hardening, and chaos engineering.

Benoit Baudry is a Professor of Software Technology at KTH Royal Institute of Technology in Stockholm, Sweden. He received his Ph.D. in 2003 from the University of Rennes and was a research scientist at INRIA from 2004 to 2017. His research is in the area of software testing, code analysis, and automatic diversification. He has led the largest research group in software engineering at INRIA, as well as collaborative projects funded by the European Union, and software companies.