A Large-Scale Empirical Study on Java Library Migrations: Prevalence, Trends, and Rationales

Anonymous Author(s)

ABSTRACT

With the rise of open-source software and package hosting platforms, reusing 3rd-party libraries has become a common practice. Due to risks including security vulnerabilities, lack of maintenance, unexpected failures, and license issues, a project may completely remove a used library and replace it with another library, which we call *library migration*. Despite substantial research on dependency management, the understanding of how and why library migrations occur is still lacking. Achieving this understanding may help practitioners optimize their library selection criteria, develop automated approaches to monitor dependencies, and provide migration suggestions for their libraries or software projects. In this paper, through a fine-grained commit-level analysis of 19,652 Java GitHub projects, we extract the largest migration dataset to-date (1,194 migration rules, 3,163 migration commits). We show that 8,065 projects having at least one library removal and 1,564 (lower-bound) to 5,004 (upper-bound) projects have at least one migration, indicating the prevalence of library migrations. We find that projects with library removals have one removal per 139 commits, and projects with migrations have 2 to 4 migrations in median. We discover that library migrations are dominated by several domains presenting a long tail distribution. Also, migrations are highly unidirectional in that libraries are either mostly abandoned or mostly chosen in our project corpus. A thematic analysis on related commit messages, issues, and pull requests identifies 14 frequently mentioned migration reasons, 7 of which are not discussed in previous work. Our findings can be operationalized into actionable insights for package hosting platforms, project maintainers, and library developers.

1 INTRODUCTION

Reusing existing 3rd-party libraries[1] with ready-to-use features has long been a common practice in software development, which can increase software quality and development productivity [52]. Still, it is not until the rise of open-source software and the availability of central hosting platforms (e.g. Maven [53], NPM [56], PyPI [32], etc) in the recent decade that software reuse becomes so easy for both library users and library developers. The number of newly published JARs on Maven Central has grown from 86,161 in 2010 to 364,218 in 2015 and 1,435,600 in 2020 [53]. As a result, 3rd-party libraries are widely adopted in both open-source and proprietary software projects, and a non-trivial software project often reuses dozens of existing libraries [78, 81].

[1]Developers use different terms interchangeably, such as libraries, packages, dependencies, components, frameworks, etc, to refer to a piece of reusable software (e.g. a JAR file [58]) available for download in a central hosting platform (e.g. Maven Central [53]). For consistency, we use the term "library" to refer to the reusable software itself, and the term "dependency" to emphasize that it is used by another software project.

ESEC/FSE '21, August 23–27, 2021, Athens, Greece
2021. ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Meanwhile, the adoption of 3rd-party libraries brings unique challenges in the entire software life-cycle [21]. First, given the wide spectrum of available libraries, even the task of choosing the right library for a specific purpose becomes non-trivial in which complex socio-technical factors need to be considered [47, 59]. Second, increasing concerns are being raised about the risk of using outdated libraries as they may contain security vulnerabilities and unresolved issues [1, 26, 60, 87], challenging the conventional strategy of "if it ain't broke, don't fix it." Consequently, researchers study the characteristics, reasons, and drivers of library updates [7, 44, 85] and practitioners propose industry solutions (e.g. Synk [48], WhiteSource [67], and GitHub Advisories [35]) that aim to keep libraries up-to-date and vulnerability-free. However, failures or misalignment with a library inevitably happens and may not be resolvable by updating its version. In such cases, the library has to be completely removed and replaced by another library, which are called *library migration* in the literature [37, 38, 71, 73].

While a large body of existing researches are conducted to characterize and understand library adoption [23, 24, 40, 45, 47, 50, 59, 82, 84], and library updates [7, 15, 20, 25, 29, 43, 44, 51, 68, 85, 87], researches on library migrations [2, 5, 6, 38, 71, 73] are still fragmented and incomplete. More specifically, we lack understanding on: 1) how a large number of projects migrate their dependencies as a whole, and 2) what factors drive such migrations. As the ultimate consequence of library adoption failures, such understanding will be a valuable source of information for developers, decision makers, and stakeholders. It may help them optimize their library selection criteria, develop automated approaches to monitor dependencies, and provide migration suggestions for their libraries or software projects. To bridge this knowledge gap, we seek to conduct a descriptive mixed methods study on large-scale open-source data to holistically understand how and why library migrations happen. More specifically, we ask the following research questions:

- **RQ1:** How common are library migrations?
- **RQ2:** How do migrations happen between libraries?
- **RQ3:** What are the frequently mentioned reasons by developers when they conduct a library migration?

We face several challenges when answering these research questions. First of all, the parallel and distributed nature of `git`-based development presents unique peculiarities [9] in an attempt to reconstruct accurate library change histories for a specific project. Second, it is difficult to precisely define "library migration" and identify library migrations in commit history. To address these challenges, we propose an event-based dependency change model, retrieve migrations using a state-of-art mining algorithm [37], and analyze migration frequency using upper-bound and lower-bound estimations. By applying our method on 19,652 Java projects and 4,022 libraries, we extract 2,629,992 dependency changes and 3,163 migration commits (the largest dataset compared with previous works in Table 1). We conduct exploratory data analysis for RQ1-2,

Table 1: Summary of Related Work on Library Migration

Paper	Subject of Study	Reported Prevalence	Reported Reasons
Teyton et al. [71]	80 migration rules	N.A.	convenience, outdated, incompatibilities
Teyton et al. [73]	329 migration rules, 1,198 migrations, 26 commit messages	5.57% of 15,168 projects	feature, performance, configuration, bug, environment
Kabinna et al. [38]	49 logging library migration attempts (33 successful)	33 of 223 ASF projects (14.80%)	feature, performance, flexibility, reduce future maintenance, reduce dependencies

and apply thematic analysis [11, 22] to migration-related commits, issues, and pull requests to answer RQ3. The key findings are:

(1) Both library removals and migrations are prevalent in the 19,652 Java projects, in which 8,065 have at least one library removal and 1,564 (lower-bound) to 5,004 (upper-bound) have at least one library migration. They are more likely to happen among projects with larger number of commits and dependencies. A median project with library removals has one removal per 139 commits and a median project with migrations has 2 to 4 migrations in total.
(2) Library migrations from four domains (logging, testing, JSON, and web service dominate the dataset, presenting a long tail distribution. And, migrations are highly unidirectional in that libraries within the same domain are either mostly abandoned or mostly adopted in the studied projects.
(3) Projects conduct library migrations for 14 different library-side and project-specific reasons. We identify 7 reasons not discussed in previous works. The most frequent reasons include lack of maintenance, feature, usability, integration with project context, and simplification of dependencies.

Based on our findings, we summarize actionable insights for package hosting platforms, project maintainers, and library developers, including how to formulate and publicize best practices, how to select and integrate libraries, what to do with unmaintained libraries, and what to prioritize for library development. We also identify aspects where current tooling support does not suffice and suggest future research directions.

The source code and data are available at https://github.com/Asdf0717/LibraryMigration, and will be archived upon acceptance.

2 BACKGROUND AND RELATED WORK

Migration is a common phenomenon during software maintenance, which may refer to different development activities that stem from various motivations. Common cases of migration include: migrating from one version to another version [19, 44], one API to another API [4, 46], one programming language to another programming language [54, 86], one platform to another platform [31, 80], or one library to another library [37, 38, 71, 73]. In this paper, we use the term **library migration** to refer to the process of replacing one library with *another*[2] functionally similar or equivalent library. Given a library migration from library l_1 to library l_2, we refer to l_1 as the **source library**, l_2 as the **target library**, and $\langle l_1, l_2 \rangle$ as a **migration rule** in the subsequent paper.

Three steps are typically required for a library migration: justifying the necessity of a migration, finding the best target library, and modifying the code to use the new library. For open-source projects, the first two steps are often facilitated through public discussions in issue trackers, where the benefits and costs are evaluated by developers [38]. Such discussions may not result in a migration if no consensus is reached or the perceived benefits do not outweigh the costs [38]. The cost mainly comes from the third step, which is known to be tedious, error-prone, and sometimes difficult [4, 13]. There are efforts that aims to improve development efficiency for the third step, by using API wrappers [5, 6], mining API mappings [3, 4, 13, 72], or directly editing code to use the new API [17, 83]. Studies also show that library migrations may improve code quality [2], but performance is rarely improved [38].

In this paper, we focus on understanding the first two steps of library migration, following existing studies [38, 71, 73] (Table 1). Teyton et al. [71] propose the concept of migration graph and a method to mine migration graph from software releases, where they get 80 migration rules and three reasons from real-world examples. In their subsequent study [73], they use a modified approach on the commit history of 15,168 projects and get 329 migration rules, 1,198 migrations and 26 commit messages that mentioned migration reasons. Kabinna et al. [38] analyze developer discussions of logging library migrations in Apache Software Foundation (ASF) projects, where they identify 49 migration attempts and five major reasons. However, the limited number of migrations obtained by Teyton et al. [71, 73] prevents them from deepening their results, while the results of Kabinna et al. [38] may not generalize to other libraries. Their limitations motivate us to conduct a new large-scale mining study to provide a comprehensive overview of how and why library migrations happen among a large number of projects and libraries.

Our work is also partially inspired by recent studies on library adoption [23, 24, 40, 45, 47, 50, 59, 82, 84] and library updates [7, 15, 20, 25, 29, 43, 44, 51, 68, 85, 87]. In terms of library adoption, researchers use interviews and surveys to summarize developer considerations when choosing JavaScript frameworks [59], deciding whether to re-implement [82], and selecting libraries in general [47]. Researchers also employ different modeling approaches to identify factors that lead to adoption between R dataframe libraries [50] and between JavaScript CI/CD tools [40, 45, 84]. In terms of library update, researchers have investigated how developers update [7] or not update libraries [20, 25, 43, 44, 68, 85], mechanisms to support library update [29, 51], and the impact of security vulnerabilities introduced by outdated libraries [26, 87]. However, they do not consider migrations between *different* libraries, and our study can complement existing researches by providing findings about how previous library adoptions fail in projects and why developers choose to replace a library instead of updating it.

[2]More precisely, by *another* we mean the *names* (i.e., `groupId:artifactId` in Java) of the two libraries are different.

3 DATA COLLECTION

3.1 Collecting Projects and Libraries

We begin with the latest Libraries.io dataset [39] (released in Jan 2020), which is widely used in related research (e.g. [1, 27, 37, 85]). We choose to focus on Java projects because of Java's popularity and industrial importance, and because all previous works in this topic focus on Java [38, 71, 73]. We consider a repository on GitHub as a project and use the GitHub repository list in the dataset to select relevant repositories. As a simple threshold for ensuring the quality of selected repositories, we select non-fork Java repositories with more than 10 stars, leaving us with 59,475 repositories. We do not filter by the number of commits and recent activities because the goal of our study is to depict general longitudinal trends, not to observe the state-of-art practice by mature projects. Then, to retrieve version control data for these projects, we use the World of Code database [49] (version R, constructed in April 2020). To simplify the task of dependency extraction, we focus on projects that use Maven [33] for build and dependency management. These projects contain one or several configuration files named `pom.xml` in their project paths, which has a `<dependencies>` section where developers declare the group IDs, artifact IDs, and version numbers of required libraries. The declarations are strict in that a build attempt will fail if a used library is not declared or included as a transitive dependency, but an unused library may still be declared, which is a threat to validity (Section 7.2). By keeping repositories with at least one `pom.xml` file in one of the repository's commits, we retain 19,652 repositories (23,988,437 commits in total).

The dependencies declared in project `pom.xml` files may not fit the definition of a "library," because they may be internal project modules not intended for reuse by other projects. To filter out such cases, we only consider dependencies that 1) are accessible in Maven Central [53], and 2) have been included in the `pom.xml`s of more than 10 repositories in the aforementioned 19,652 repositories. Finally, we get 4,022 libraries[3]. We sample 94 libraries (confidence level = 95%, confidence interval = 10) and determine whether they are libraries or not by inspecting their descriptions in Maven Central and searching them on the Web. Ninety-three of 94 samples (98.93%) contain public information indicating their appropriateness of reuse (e.g., terms such as library, framework, platform, specification, etc), which justifies our selection criteria.[4]

3.2 Computing Dependency Changes

With projects and libraries collected, our next task is to extract dependency changes from project commit histories, as these dependency changes may indicate library migrations. However, simply sorting and comparing all versions by time for a given `pom.xml` file will generate too many false positives because a project may have many parallel branches which may or may not be merged, effectively forming a directed acyclic graph [9] (DAG, see Figure 1). To take the DAG into consideration, we model each **dependency change** as an *event* happened in one commit, which can be either an adoption, a removal, or a version change. The event is computed

[3]Following general conventions, we consider a unique (group ID, artifact ID) pair as a library, and the version string for distinguishing different versions of a library.

[4]The only exception is `io.prometheus:simpleclient_common` which describes itself as *common code used by various modules of the* `simpleclient`.

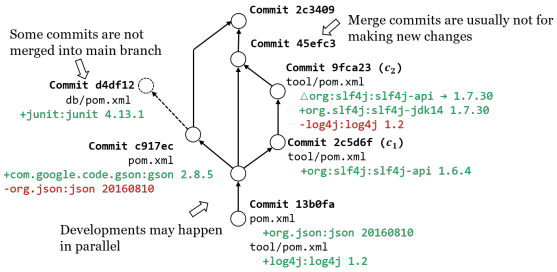


Figure 1: An example project commit history.

Table 2: Statistics of the Collected Dataset

Data	Size/Statistics
Projects ($\mathcal{P}$)	19,652
Stars	median = 32, stddev = 1407.93
Commits	median = 142, stddev = 5614.70
Active Months	median = 27, stddev = 26.97
`pom.xml` Files	median = 2, stddev = 75.72
Libraries ($\mathcal{L}$)	4,022
Versions	median = 26, stddev = 104.20
Included Repositories	median = 25, stddev = 310.94
Dependency Changes ($\mathcal{D}$)	2,629,992 (302,774 commits)
Adoptions ($\mathcal{D}^+$)	1,771,664 (208,691 commits)
Removals ($\mathcal{D}^-$)	290,150 (56,377 commits)
Version Changes ($\mathcal{D}^v$)	568,178 (97,447 commits)
Migration Rules ($\mathcal{R}$)	1,194 (393 renames)
Migration Commits ($\mathcal{M}$)	3,163 (1,459 renames)

from comparing with the prior `pom.xml` file version in the parent commit. More precisely, let commit c_1 be the parent of commit c_2, L_1 be the set of dependencies in `pom.xml` file f of commit c_1, and L_2 in file f of commit c_2. For c_2, the set of adoptions $D^+_{c_2} = L_2 - L_1$; the set of removals $D^-_{c_2} = L_1 - L_2$; and the set of version changes $D^v_{c_2} = \{l \mid l \in L_1 \wedge l \in L_2 \wedge ver(l, c_1) \neq ver(l, c_2)\}$. For c_2 in Figure 1, f = `tool/pom.xml`, $D^+_{c_2}$ = {`org.slf4j:slf4j-jdk14`}, $D^-_{c_2}$ = {`log4j:log4j`}, and $D^v_{c_2}$ = {`org.slf4j:slf4j-api`}. We ignore merge commits based on the assumption that merge commits are not used for making new changes other than conflict resolution.[5] Through an iteration over all commit diffs of all projects, we get 2,629,992 dependency changes from 302,774 commits.

3.3 Identifying Library Migrations

The dependency changes we collected before may not be necessarily related to a library migration, but it is non-trivial to define and identify real library migrations from dependency changes. First, we need to determine whether a migration is feasible between two arbitrary library pairs (i.e., for l_1 and l_2, whether $\langle l_1, l_2 \rangle$ is a migration rule). Next, we need to identify commits related to a library migration. To deal with these challenges, we define a commit as a **migration commit** when its commit message clearly indicates a migration (e.g. *Replace org.json with jackson* [28]), and define $\langle l_1, l_2 \rangle$ as a **migration rule** if and only if we can find a migration commit that conducted the migration from l_1 to l_2.

[5]This assumption is widely used. For example, GitHub also excludes merge commits when counting the number of contributions and changed lines that a user made.

We acquire migration commits and migration rules from the dependency changes using a state-of-art mining algorithm [37] and manually validate the returned results. For each source library query, the algorithm mines possible target libraries, rank each target library by a confidence value, and returns possible migration commits from the source library to the target library based on the collected dependency changes. However, it generates a tremendous amount of output if we query using all the 4,022 libraries. To reduce human inspection effort, we use a combination of 1) source libraries provided in [71], 2) 500 most popular libraries by number of times added in our repository dataset, as the query to the algorithm (670 queries in total). For the output of each query, we focus on inspecting the returned commits for target libraries that have non-zero confidence value, rank top-20 among all returned target libraries, and exist in the aforementioned 4,022 libraries. After validation, we obtain 1,194 migration rules (390 source libraries, 562 target libraries) and 3,163 migration commits. We obtain the largest dataset compared with previous work (Table 1), but the size of migration commits by this definition is doomed to be small because most commits do not have informative commit messages [18]. Thus, this set of migrations can only be viewed as a *subset* of real world migrations and inadequate for answering migration popularity (RQ1), but they are guaranteed to be correct and invaluable for unveiling migration behavior (RQ2) and reasons for migration (RQ3).

Some migrations happen because of a library *rename*, which often accompanies important library changes (e.g., major version update, license change, organization switch, etc) and falls near the boundary of library update and library migration. We detect these rules by finding all rules whose artifact IDs contain overlapping terms (excluding terms like `api`, `core`, `all`, etc), and manually validating all the detected results, resulting in 393 **rename rules**. In subsequent analysis, we distinguish rename rules from other migration rules where their property significantly differs.

In the remainder of this paper, we denote $\mathcal{P}$ as the set of repositories (i.e., projects), $\mathcal{L}$ as the set of libraries, $\mathcal{D}$ as the set of dependency changes ($\mathcal{D}^+$, $\mathcal{D}^-$, $\mathcal{D}^v$ for adoption, removal and version changes, respectively), $\mathcal{R}$ as the set of migration rules, and $\mathcal{M}$ as the set of migration commits. The statistics of collected data are summarized in Table 2.

4 RQ1: HOW COMMON ARE LIBRARY MIGRATIONS?

Since library migration is the process of removing a library and introducing another library, a library removal and a library adoption are *necessary* for a library migration. To the best of our knowledge, library removals during software evolution have never been studied before. Intuitively, if many projects remove a library, it is probably followed by migrations, and it is unknown exactly how many removals are related to migrations and how migrations are distributed in our project corpus. Thus, we ask two sub-RQs:

RQ1.1: How frequently do projects remove a library?

RQ1.2: How frequently do projects migrate a library?

4.1 RQ1.1: Removal Frequency Analysis

4.1.1 Methodology. We consider all types of dependency changes during analysis for comparison purposes. For project $p \in \mathcal{P}$, let C_p be the set of commits for p, D_p be the set of dependency changes happened in p (i.e., $D_p = \bigcup_{c \in C_p} D_c$). We first consider projects $P_r \subset \mathcal{P}$ with at least one removal, i.e., $P_r = \{p \mid p \in \mathcal{P} \wedge D_p^- \neq \emptyset\}$, and analyze its distribution within $\mathcal{P}$. Then, for $p \in P_r$, we compute and compare the distributions for 1) the number of changes in project p for all change types (i.e., $\sum_{c \in C_p} |D|$ for $D \in \{D_c, D_c^+, D_c^-, D_c^v\}$), and 2) the average number of commits between changes for all change types (i.e., $|C_p|/|\{c \mid c \in C_p \wedge D \neq \emptyset\}|$ for $D \in \{D_c, D_c^+, D_c^-, D_c^v\}$). We use the latter to estimate the frequency (or interval) of each change type along the project development history.

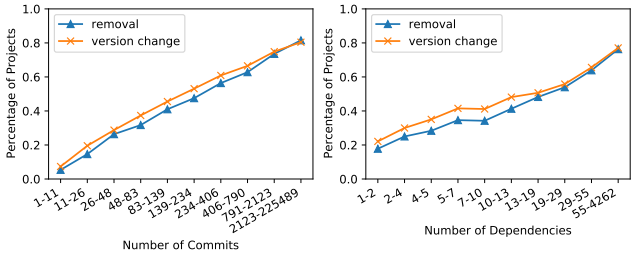


Figure 2: Distribution of P_r by number of commits / dependencies, with that of $P_v = \{p | p \in \mathcal{P} \wedge D_p^v \neq \emptyset\}$ for comparison.

4.1.2 Results. For all 19,652 projects ($\mathcal{P}$), 1398 (7.11%) projects do not have any dependency changes because their `pom.xml` files have no or empty `<dependencies>` sections. Among the remaining 18,254 projects, 8657 (47.43%) have at least one version change and 8045 (44.07%) have at least one removal. The two major reasons for not having any removals or version changes are that, these projects do not have sufficient development histories or only have a few number of dependencies. By dividing the 18,254 projects into 10 equally sized chunks separated by number of commits, we show on the left side of Figure 2 that migrations are more likely to happen in projects with higher number of commits, and the trend is very similar to that of version changes. A Spearman correlation test between number of removals and number of commits for all projects yields a coefficient ρ of 0.532 ($p < 0.001$), which indicates moderate to strong correlation. A similar trend can be observed for number of dependencies on the right side of Figure 2, but the correlation is weaker ($\rho = 0.158$, $p < 0.001$).

For the 8,045 projects with at least one removal (P_r), we plot the distribution of the number of changes and the average number of commits between changes for all change types in Figure 3. Among P_r, a median project has 35 adoptions, 6 removals, 6 version changes, one adoption per 48 commits, one removal per 139 commits, and one version change per 167 commits. The variations are also high across different projects, with version changes having higher variance than removals. We conclude that apart from library updates and adoptions, library removals are also very common for these projects.

Summary for RQ1.1:

Among projects with dependencies, 44.07% have at least one library removal. For those projects, a median project has one removal per 139 commits (for comparison, one version change per 167 commits). Removals are more likely to happen for projects with higher number of commits and dependencies.

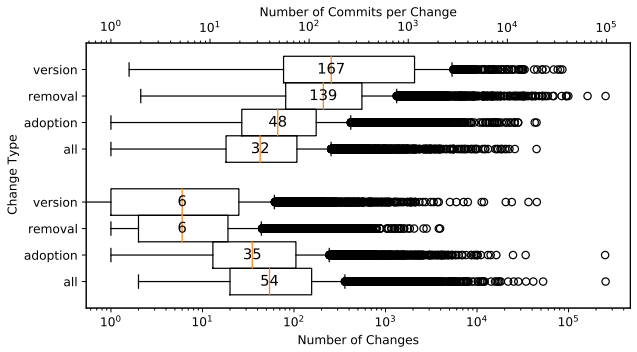


Figure 3: Distribution of the number of changes (below) and the average number of commits between changes (above).

4.2 RQ1.2: Migration Frequency Analysis

4.2.1 Methodology. We begin with $\mathcal{M}$ and $\mathcal{R}$ mentioned in Section 3.3, which contains 716 libraries (denote as $L_m \subset \mathcal{L}$). By limiting our analysis on projects that have once adopted any of these libraries, we obtain 17,426 projects and denote as $P_m = \{p \mid p \in \mathcal{P} \wedge \exists l,\ l \in L_m \wedge l \in D^+_p\}$.

The computation of frequency for library migrations is difficult because we cannot guarantee the completeness of mined migrations for any non-trivial mining approach [37, 71]. As mentioned in Section 3.3, $\mathcal{M}$ and $\mathcal{R}$ are subsets of real world migration commits and migration rules, which can only be used to estimate lower-bound frequencies. We refer to such cases as **confirmed migrations** in this section and denote $P_{cm} \subset \mathcal{P}$ as the set of projects with at least one confirmed migration. As an upper-bound estimation, we propose it is likely that p has also conducted the same migration if $\langle l_1, l_2\rangle$ is a migration rule and l_1 is removed and l_2 is added in a `pom.xml` file of project p. More formally, let $D_{p,f}$ be the set of dependency changes happened in `pom.xml` file f of project p, we hypothesize that if $\langle l_1, l_2\rangle \in \mathcal{R} \wedge l_1 \in D^-_{p,f} \wedge l_2 \in D^+_{p,f}$, a library migration may have happened in f of p. Similar to P_{cm}, we refer to such cases as **possible migrations** and denote $P_{pm} \subset \mathcal{P}$ as the set of projects with at least one possible migration.

Similar to RQ1, we first analyze the distribution of P_{cm} and P_{pm} within P_m and its relationship with number of commits and number of dependencies. Then, we analyze the distribution of library migrations within P_{cm} and P_{pm}. For each project p, we count the number of migrations by the number of migration rules to avoid duplication when a project simultaneously performs the same migration in many `pom.xml` files, or in different branches in which some may not be merged. More formally, we define

$$MigCnt(p) = |\{\langle l_1, l_2\rangle \mid \langle l_1, l_2\rangle \in \mathcal{R} \wedge l_1 \in D^-_{p,f} \wedge l_2 \in D^+_{p,f}\}| \quad (1)$$

4.2.2 Results. Among the 17,426 projects (P_m), 7,950 (45.62%) have at least one removal of libraries in L_m, 5,004 (28.72%) projects have at least one possible migration (P_{pm}), and 1,564 (8.98%) projects have at least one confirmed migration (P_{cm}). As a rough estimation, up to 62.94% (5,004 / 7,950) of the projects have removals that lead to a library migration, for 31.25% (1,564 / 5,004) of which we can confirm in commit messages. For migration frequency, even the lower-bound estimation of 8.98% is higher than the reported prevalence in Teyton et al. [73] (see Table 1), indicating that library

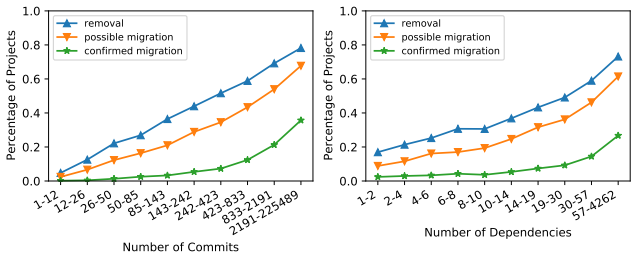


Figure 4: Distribution of P_{cm} and P_{pm} by number of commits and dependencies. We also show the results of $P'_r = \{p | p \in P_m \wedge D^-_p \neq \emptyset\}$ for comparison.

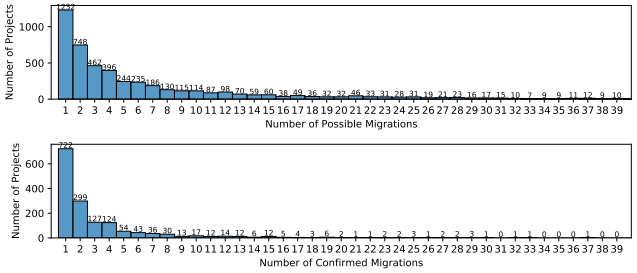


Figure 5: Distribution of projects by number of migrations.

migrations have become more prevalent since their publication. Although library migrations may not be very common in P_m which contains projects with few commits or few dependencies, we show that library migrations are much more common for projects with many commits and dependencies (Figure 4) ($\rho = 0.456$, $p < 0.001$ for commits, and $\rho = 0.075$, $p < 0.001$ for dependencies using Spearman correlation test). Among the 3,796 projects with more than 10 dependencies and 143 commits (both are median values in P_m), 2161 (56.93%) / 764 (20.13%) projects have at least one possible / confirmed migration. Finally, we plot the distribution of projects by number of migrations happened in P_{pm} and P_{cm} (Figure 5). As expected, their distributions resemble the shape of a long-tail distribution, where most projects have one or a few migrations and some projects have many more. A median project in P_{pm} has undergone 4 possible migrations (mean = 9.33, stddev = 17.30, max = 337) and a median project in P_{cm} has undergone 2 confirmed migrations (mean = 3.35, stddev = 4.54, max = 46). Although there are extreme cases, the majority of projects (61.59%~84.78%) have no more than five migrations.

Summary for RQ1.2:

In the studied project corpus, 8.98% to 28.72% have undergone at least one library migration. For those projects, most have no more than five migrations. The probability of undergoing a library migration becomes significantly higher for projects with larger number of commits and dependencies.

5 RQ2: HOW DO MIGRATIONS HAPPEN BETWEEN LIBRARIES?

The results of RQ1 have demonstrated that library migrations have happened in a significant number of software projects. However, it is still unclear how exactly library migrations are conducted, in

particular, what kind of migrations are performed, among what kind of libraries, and whether or not the migrations share common behavior. Therefore, we ask **RQ2:** How do migrations happen between libraries?

5.1 Methodology

We adopt the concept of **migration graph** proposed by Teyton et al. [71, 73] as the basis to facilitate further analysis. We define the migration graph in our study as a weighted directed graph $\langle L_m, \mathcal{R}, w \rangle$, where libraries in L_m are nodes, migration rules in $\mathcal{R}$ are edges, and $w()$ is an edge weighting function. For $\langle l_1, l_2 \rangle \in \mathcal{R}$, we compute weight $w(l_1, l_2)$ by counting the number of projects that have performed a migration from l_1 to l_2 in $\mathcal{M}$.

To demonstrate the extent to which a library is more likely to be a migration source or a migration target, we define the following metric $flow(l)$ for library $l \in L_m$

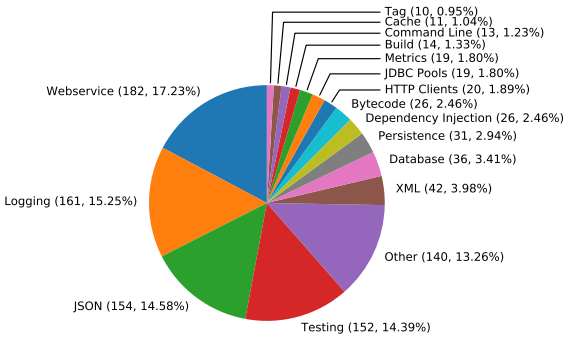
$$flow(l) = \frac{deg^-(l) - deg^+(l)}{deg^-(l) + deg^+(l)} \quad (2)$$

where $deg^-(l)$ is the weighted indegree for node l and $deg^+(l)$ is the weighted outdegree for node l. Obviously, $flow(l) \in [-1, 1]$, where $flow(l) = 1$ means l is always a target library and $flow(l) = -1$ means l is always a source library.

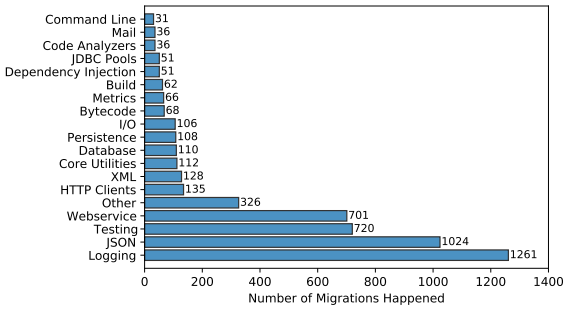
To understand what kind of migrations are performed and to effectively visualize the graph, we divide migration rules into different application domains. First, we use Louvain community detection [10] on the migration graph to get 116 initial clusters. Then, we use library descriptions, labels in Maven Central, and information from search engines to merge clusters, resolve cluster names, and correct errors. The application domain of each migration rule (l_1, l_2) is assigned from the domain of l_1 and l_2 when the two libraries are in the same domain, which is the case for 1,055 rules (88.44%). For the remaining rules, l_1 and l_2 fall into different domains either because the boundaries between domains is sometimes unclear or migrations can happen between different domains (e.g. from a XML library to a JSON library for serialization). We get 53 application domains in total, among which 16 domains have more than 10 migration rules. The domain distribution is shown in Figure 6a. Four domains (web service, logging, JSON, and testing) dominate the dataset with 648 migration rules (61.42%). If we count the number of migrations happened (Figure 6b), migrations between logging libraries and JSON libraries happen 1.4-1.7x more than testing and web service libraries. Finally, we visualize sub-graphs for important application domains using Sankey diagrams [64], which are often used to visualize how things flow in complex networks.

5.2 Results

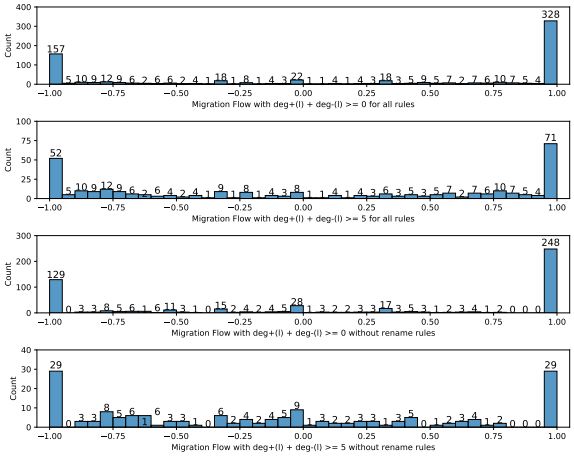
Figure 6c (top) shows the distribution of $flow(l)$ for $l \in L_m$. Surprisingly, the distribution is extremely distorted. For 154 libraries (21.51%), $flow(l) = -1$ and for 326 libraries (45.53%), $flow(l) = 1$. The remaining libraries generate a uniform distribution in the interval of $(-1, 1)$. The distribution shape does not change even if we only consider libraries with higher indegree and outdegree values (Figure 6c second top), although there are slightly more libraries that fall within the interval of $(1, -0.75] \cup [0.75, 1)$. The distribution shape also does not change even if rename rules are excluded (Figure 6c second bottom and bottom). We can conclude from this



(a) Distribution of $\mathcal{R}$ by application domain.



(b) Number of migrations happened for migration rules in each application domain ($\sum_{p \in \mathcal{P}} MigCnt(p)$, Equation 1).



(c) Distribution of $flow(l)$ under different conditions.

Figure 6: How migrations happen between libraries.

observation that a large number of libraries are either mostly abandoned or mostly adopted as migration targets by developers, which indicates that some libraries are more competitive than others and should be recommended to use instead of the other libraries.

As illustrating examples, we visualize the sub-graphs of three most popular application domains (logging, testing, and JSON) in Figure 7. We exclude the sub-graph for web service due to space constraints. To make the graphs readable, we only retain migration rules $\langle l_1, l_2 \rangle$ with $w(l_1, l_2) \geq 10$ for testing and JSON, and

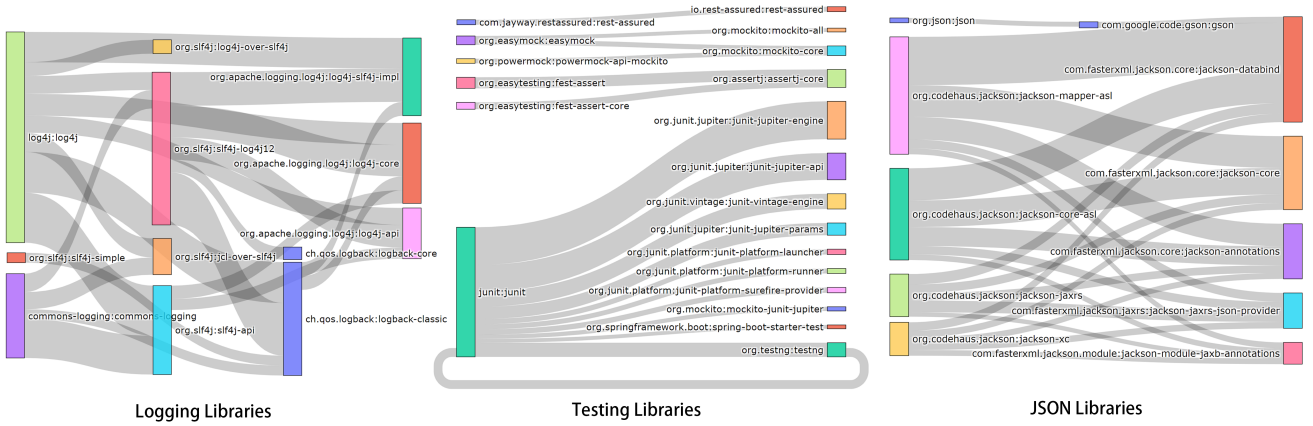


Figure 7: The migration sub-graphs for logging, testing, and JSON libraries. Migrations flow from left to right. Each rectangle is a graph node (i.e, library) and each grey band is a graph edge (i.e., migration rule). For each edge, its band width encodes the number of repositories where migrations happened for this migration rule.

$w(l_1, l_2) \geq 15$ for logging. The unidirectional trend is clearly visible using Sankey diagrams. In the case of logging libraries, `commons-logging` [75] and `log4j` [74] are two mostly abandoned libraries, while libraries under group ID `org.apache.logging.log4j` (i.e., `log4j 2` [76]) are mostly adopted. Libraries related to `slf4j` [63] and `logback` [62] are somewhere in between. The reason for such phenomenon may be that the Java community has two major logging practice changes [38]: one is from ad-hoc logging libraries (e.g., `log4j`) to logging abstraction libraries (e.g., `slf4j`); another is from logging abstraction libraries to logging unification libraries (e.g., `log4j 2`). Such multiple paradigm shifts can also be observed in JSON libraries (e.g., `gson` [36] is first adopted but then switched to `jackson` [30]), but not obvious in testing libraries. We can also see that rename rules constitute a large number of migrations for testing and JSON (many non-rename rules are omitted in Figure 7 because they happen in less than 10 projects). In all cases, hardly can we see two libraries with major bi-directional flows, except the case of `junit:junit` [77] and `org.testng:testng` [8], which may be the results of early competition when both libraries are immature.

Summary for RQ2:

Library migrations from four domains (logging, testing, JSON, and web service) among 53 dominate the dataset, presenting a long tail distribution. Also, library migrations are highly unidirectional in that most libraries are either mostly adopted or mostly abandoned.

6 RQ3: WHAT ARE THE FREQUENTLY MENTIONED REASONS BY DEVELOPERS WHEN THEY CONDUCT A LIBRARY MIGRATION?

RQ1 and RQ2 demonstrate *how* library migrations happen in Java software projects, but it is still unclear *why* library migrations happen in these projects. Understanding the latter is important because we can extract retrospective insights for practitioners by summarizing the rationale behind previous migrations. Therefore, we ask

RQ3: What are the frequently mentioned reasons by developers when they conduct a library migration?

6.1 Methodology

To answer RQ3, we choose to study developer-written texts related to existing migrations. We begin with the commit messages of the 3,340 migration commits ($\mathcal{M}$, Section 3.3). To extend available data, we follow GitHub conventions to identify issue references in commit messages [34], and retrieve pull requests containing the commits using the GitHub API. We get 2,775 pull requests and 385 issues after this step. To extract reasons from these related texts, we follow the general procedure of thematic analysis [11, 22], which is frequently used to extract recurring patterns from text in software engineering research (e.g., [14, 66, 70]).

Since a large amount of text in commit messages, issue/PR descriptions and comments are not relevant to migration reasons, three of the authors first independently read and re-read all text in the commits, issues and PRs to mark relevant text about migration reasons. After the marks are merged and verified through discussion, we get 351 (10.51%) commits, 223 (8.04%) pull requests and 112 (33.53%) issues that include some text stating the reasons for this migration. The ratios are low either because developers just document what they have done instead of the rationale behind their work or because some issues and PRs are discussing other topics, but we still acquire the largest amount of data to answer this question compared with existing works [38, 73] (Table 1).

Next, one author reads through all materials and generates 23 initial codes for migration reasons and a coding book that defines these codes. Then, he and another author independently use the coding book to code all materials, in which each commit/issue/PR can be assigned multiple codes. To deal with materials with more than one code, we use MASI distance [61] as the distance measure and Krippendorff's alpha [41] to measure inter-rater agreement. The first round of coding results in a Krippendorff's alpha of 0.731, which is lower than the recommended threshold of 0.8 [42]. To reduce disagreement, we identify ambiguous codes and discuss solutions until a consensus is reached. We decide to merge some of the ambiguous codes (e.g., "ease of use" and "flexibility" are merged

into "usability") to generate final themes, and resolve the remaining conflicts after the merging. Finally, the 23 codes are merged into 14 sub-themes organized into three themes (source library, target library, and project specific) and an additional "other" theme. The Krippendorff's alpha of initial coding results after mapping to the 14 sub-themes is 0.803. The main reason for most disagreements is that some text do not provide sufficient information for distinguishing between different sub-themes, and the resolution process inevitably includes searching over the Web and the project development history, and inferring the underlying reasons based on our prior experience (one author has 4 years of Java development experience and arbitrates most of the cases). There are still 8 cases where the conflicts cannot be resolved because the text is ambiguous or not understandable, and we simply label them as "other."

6.2 Results

We summarize 14 reasons for migration in Table 3 with three themes: 1) reasons originated from the *source library*, 2) reasons motivated by the *target library*, and 3) *project specific* reasons. To avoid duplication (i.e., multiple issues/PRs mentioning the same migration), we estimate the frequency of each reason by the number of projects that have a commit/issue/PR that mentioned this reason (420 projects in total). We also show the proportion of rename rules among all rules and top-4 application domains by the aforementioned frequency for each reason. Among the 14 reasons, 7 reasons (marked with †) are not discussed in existing work (Table 1), and 5 reasons (marked with §) originate from existing work but are further extended and re-organized based on our findings. In the remainder of this section, we will introduce each reason in detail.

6.2.1 Source Library. 135 projects (32.15%) conduct library migrations because of problems in the source library, with three dominating factors: the source library is not maintained/outdated, suffers from security vulnerabilities, or contains bugs/other issues.

Not Maintained/Outdated. The migration happens because the source library is old and/or lacks maintenance, and the project decides to use the library that is currently recommended and supported by the community. It is the most common reason in source libraries for both rename rules and other rules. Examples: 1) *Replace end of life apache http client 3.1 with 4.1.*[6] 2) *FEST hasn't been updated in over 1 year. AssertJ looks like the successor.*[7] Teyton et al. [71] mention "outdated" as a reason for migration, but we further discover that the major concern is whether the library is still maintained, and the recognition of non-maintenance (e.g., announcement of end-of-life) often triggers migrations.

Bug/Other Issues. The migration happens because of bugs, warnings, crashes, inconsistencies, unwanted features or other peculiar issues. For example, *Empty string is not working, when we pass ""...This seems to be a bug in jcommander.*[8]

Security Vulnerability. The migration happens because of security vulnerabilities in the source library, and also mostly because the source library does not provide any security fixes. This happens mostly for rename rules (67.50%), and most notably for some of the migrations from `log4j` to `log4j 2`, `slf4j` or `logback`, e.g. *Fix for log4j vulnerability CVE-2019-17571.*[9] It seems that security issue is the last straw that pushes these projects to finally abandon an old and unmaintained library.

6.2.2 Target Library. 166 projects (39.52%) conduct library migrations because of advantages in the target library. Inspired by Larios Vargas et al. [47], we divide them into the following 7 dimensions.

Usability. The most frequently mentioned reason in target library is that the target library is easy to use, has nice APIs, results in cleaner code, cleaner configuration, or is flexible for end users to switch implementations on need, e.g. *allow users to use their preferred logging framework (such as logback)* ,[10] and *Fest is awesome. The tests read better, and fest's fluent matchers are way easier to find (via auto-complete) than the static imported hamcrest matchers.*[11] Some projects also mention that choosing an easy-to-use library is important for maintainability in the future. Kabinna et al. [38] mention "flexibility" and Teyton et al. [71] mention "convenience" as reasons for migration, but the former reason is specific to logging libraries (because many projects want to allow its downstream projects to switch underlying logger implementation on demand), and the latter term does not cover all the cases we find. Therefore, we decide to merge them into a broader sub-theme of "usability."

Feature. A large number of migrations also happen because the target library has specific features that the source library cannot provide. For example, *Use gson to preserve the order of all fields.*[12] The mentioned features are highly diverse across different application domains and project contexts.

Performance. Performance is also frequently mentioned as a reason for migration. In most cases, the migration happens because a project needs to meet special functional requirements (e.g. latency, throughput, compile time, etc) that the existing library cannot satisfy. For example, *log4j has been identified as a performance bottleneck for high qps use cases.*[13]

Size/Complexity. Some projects care about the binary size or simplicity of the library, mostly because they want their own binary to be as small as possible, e.g. *Swapped H2 database to Hypersonic SQL to make stubby4j JAR smaller.*[14]

Popularity. Surprisingly, popularity is not a frequently mentioned reason, and most of the cases are because of library renames (64.71%). It may be because it is too trivial to be mentioned or discussed, or because popularity alone does not provide immediate benefits and thus is not the decisive factor of a migration.

Stability/Maturity. A few projects mention that the target library is stable or more robust. It may not be the decisive factor, but the factor that finally drives a migration, e.g., *JUnit 5 will be available soon as a stable release. It is available a alpha/beta now. As soon as it is available, all the unit tests may use this version.*[15]

Activity. A few projects mentioned the community is more active in the target library, but it is only mentioned along with other reasons (e.g. source library is not maintained) and never mentioned as the single decisive factor.

[6] Commit `fa6f20b` in https://github.com/basho/riak-java-client
[7] https://github.com/dropwizard/dropwizard/issues/493
[8] https://github.com/ballerina-platform/ballerina-lang/issues/229
[9] https://github.com/OpenLiberty/ci.maven/pull/717
[10] https://github.com/apache/accumulo/pull/1163
[11] Commit `05263e7` in https://github.com/glowroot/glowroot
[12] https://github.com/apache/incubator-pinot/pull/2473
[13] https://github.com/apache/incubator-pinot/pull/4139
[14] Commit `ce45e04` in https://github.com/azagniotov/stubby4j
[15] https://github.com/sarl/sarl/issues/875

Table 3: Reasons for Library Migration. Reasons marked with † are not mentioned in existing work on library migrations. We also extend and re-organize reasons mentioned in existing work (marked with §).

Reason	Projects	Rename Rules	Top-4 Application Domains
Source Library	135 (32.14%)	68/135 (50.37%)	Logging (23), HTTP Clients (19), Testing (16), JSON (15)
Not Maintained/Outdated§	77 (18.33%)	25/54 (46.30%)	Testing (13), Logging (10), JSON (8), HTTP Clients (8)
Bug/Other Issue§	41 (9.76%)	35/71 (49.30%)	Logging (7), XML (5), Database (3), HTTP Clients (3)
Security Vulnerability†	26 (6.19%)	27/40 (67.50%)	HTTP Clients (9), Logging (6), JSON (5), XML (5)
Target Library	166 (39.52%)	77/247 (31.17%)	Logging (55), Testing (33), JSON (14), Web Service (11)
Usability§	76 (18.10%)	20/95 (21.05%)	Logging (35), Testing (13), Database (4), JSON (3)
Feature	57 (13.57%)	53/125 (42.40%)	Testing (13), Logging (9), JSON (8), Web Service (5)
Performance	28 (6.67%)	7/32 (21.88%)	Logging (5), Web Service (3), Database (3), HTTP Clients (2)
Size/Complexity†	10 (2.38%)	0/10 (0.00%)	Logging (2), JSON (1), Database (1), Bytecode (1)
Popularity†	9 (2.14%)	11/17 (64.71%)	Testing (3), Logging (2), HTTP Clients (1), I/O (1)
Stability/Maturity†	8 (1.90%)	3/8 (37.50%)	Logging (2), Testing (2), JDBC Pools (1), Persistence (1)
Activity†	6 (1.43%)	0/5 (0.00%)	Testing (2), Reflection (2), Classpath (2), Code Analyzer (1)
Project Specific	188 (44.76%)	77/247 (36.82%)	Logging (42), JSON (24), Testing (24), Web Service (22)
Integration§	125 (29.76%)	90/207 (43.48%)	Logging (24), Web Service (17), Testing (16), JSON (12)
Simplification§	53 (12.62%)	21/81 (25.93%)	Logging (21), Testing (9), JSON (5), Web Service (5)
License†	22 (5.24%)	4/20 (20.00%)	Code Analyzers (6), JSON (6), PDF (3), Math (2)
Organization Influence†	5 (1.19%)	9/12 (75.00%)	Command Line (2), JSON (1), Build (1), XML (1)
Other	21 (5.00%)	24/42 (57.14%)	JSON (6), Logging (6), Web Service (4), Database (2)

6.2.3 Project Specific. 188 projects (44.76%) mentioned that they conduct library migrations due to project specific reasons.

Integration. The most common reason is the need to integrate the library with something else in the project to achieve specific goals, avoid issues, or ensure compatibility. We identify three major cases: 1) to integrate with other dependencies to achieve a specific goal, e.g. utilize existing integration support, as stated in *This change replaces the internal usage of Dropwizard in favor of Micrometer...This allows to take full advantage of the Spring Boot Micrometer integration.*[16] 2) to avoid conflicts / incompatibility with other dependencies or other project modules, e.g., *Removes the PowerMock dependency which was having some bad interactions with Mockito.*[17] 3) to be compatible with the required runtime environment, e.g., *migrate from lombok to kotlin...to a fix a bug with java 11.*[18] Teyton et al. [73] mention "configuration" and "environment", which we merge and extend into this sub-theme.

Simplification. Some migrations happen to simplify library usage within project, achieve consistent style, or cleanup unnecessary dependencies, as a preventive measure to control project complexity, remove technical debt, and possibly reduce maintenance effort in the future. For example, 1) *Unify all JSON usage to fasterxml.jackson package. Remove the usage of json package from org.json, org.codehaus.jackson, com.google.code.gson, com.alibaba. Add JsonUtils class to reuse ObjectMapper and provide util methods.*[19] 2) *Make Logging Frameworks Coherent Across Libraries and IOT Codebase.*[20] 3) *Lots of cleanup. Removed Junit from a couple of modules (everything is TestNG now).* [21] This sub-theme covers "reduce the number of dependencies " and "reduce future maintenance" mentioned in Kabinna et al. [38], and we further discover a frequent pattern in which many migrations happen because developers want to avoid using different libraries for the same purpose.

License. For some projects and libraries, license compatibility stands out as the major reason for migration. For example, 6 projects, including Spring Framework and Apache Hive, refrained from using `org.json:json` because of its unusual license term *The Software shall be used for Good, not Evil.*[22] Other projects also migrate from various GPL or LGPL licensed libraries to other alternatives with more permissive licenses, such as Apache 2.0 and MIT.

Organization Influence. Some projects remove a library because it is not approved by the belonging organization, or use a library as requested by the organization, e.g. *Removes JLine2 as a dependency since it's not approved by LocationTech.*[23]

Summary for RQ3:

Projects conduct library migrations for 14 diverse reasons from the source library, the target library, and the projects themselves. The most frequent reasons are: lack of maintenance in the source, feature / usability in the target, integration with project context, and simplification of dependencies.

[16] Commit `905e384` in https://github.com/eclipse/hono/
[17] Commit `2aed070` in https://github.com/GoogleCloudPlatform/java-docs-samples/
[18] https://github.com/spotify/ffwd/pull/140
[19] https://github.com/apache/incubator-pinot/pull/3677
[20] Commit `dd5ae79` in https://github.com/hortonworks/streamline/
[21] Commit `dfe5b79` in https://github.com/dhanji/sitebricks
[22] https://www.json.org/license.html
[23] Commit `833638f` in https://github.com/locationtech/geogig/

7 DISCUSSION

7.1 Takeaways for Practitioners

7.1.1 Best Practices. Since our findings reveal that library migrations are highly unidirectional, it makes sense to formulate best practices on library migrations, and such best practices should be more publicly visible. For example, package hosting platforms can compute and demonstrate the current retention rate for a library, as an additional metric to help developers decide whether to adopt or migrate the library. Tools that help developers choose among a set of candidate libraries (e.g. AlternatveTo [57] and SimilarTech [12]) can make recommendations based on historical adoption, removal and migration trends. Tools can also be developed to monitor the "freshness" of dependencies and prompt migration recommendations to developers based on the latest trends.

7.1.2 Unmaintained Libraries. Our findings also reveal that a large part of library migrations happen because of unmaintained libraries (Table 3), and the situation significantly worsens if an unmaintained library has unfixed issues and security vulnerabilities. However, open-source libraries are prone to sustainability failures [14, 79], and unmaintained libraries may have vulnerabilities which may be even unknown to public but already discovered by malicious parties. Given such risks, we suggest that software projects should *refrain* from using an unmaintained library at all, and migrate to other alternatives as soon as possible. However, determining whether a library is unmaintained may not be easy without community effort. For example, we cannot easily tell whether a library is unmaintained from its last release time, because a mature library may not need to release frequently even if someone is still maintaining it. Therefore, we also suggest package hosting platforms build a "deprecation" mechanism for their hosted libraries (similar to that of API). Library maintainers can use this mechanism to announce their decision to no longer maintain a library, and package manager can automatically issue warnings to developers when they attempt to build their project with a "deprecated" library. A similar deprecation mechanism is available in `npm` for JavaScript libraries [16], but not in Maven Central for Java libraries.

7.1.3 Library Selection and Development. We discover that among the library selection factors reported by Larios Vargas et al. [47], only a few of them serve as the major driving pressure for library migration. Most migrations happen because of usability, feature, or performance, and different application domains have different patterns (Table 3). Given that library migrations are generally costly [4, 13], it indicates that the gain in feature, performance or usability outweighs the costs, or the migration may be mission-critical and must be performed. Therefore, we recommend developers to pay extra attention to these aspects and evaluate them with project requirements when selecting libraries, to avoid a costly migration in the future. Library developers can also consider prioritizing their efforts on these aspects to make their library more competitive.

7.1.4 Project Integration and Maintenance. A large number of library migrations happen for integration with project context, simplification of dependencies, or license incompatibilities. To prevent such migrations, we recommend project maintainers to check dependency conflicts or license incompatibilities before adopting a library, and avoid unnecessary dependencies (e.g., using different libraries for the same task). Maintainers can use automated tools to facilitate these checks (e.g., `maven-shade-plugin`, `mvn dependency` commands, and open-source license checkers). However, existing tools may be unable to check complex interaction issues or recommend best co-usage practices. For example, one developer states in an issue that *Spark really wants you to use log4j*,[24] but it is not obvious from Spark documentation. Therefore, we also suggest library developers, especially framework developers, should explicitly document what are recommended (or not) to use together with their library. Researchers may also consider developing automated tools to alleviate this issue, e.g., by applying existing library recommendation approaches [55, 65, 78] to discover co-usage patterns, or design new approaches to detect common integration pitfalls.

7.2 Threats to Validity

7.2.1 Internal Validity. One problem with Maven is that the dependencies declared may not be actually used in code, which means some removals will not be detected and we may even underestimate the number of removals and the frequency of migrations. Our approach for computing dependency changes may still generate errors in extreme corner cases (e.g., a merge commit is squashed with its parents [69]), but we believe they are very rare and the noise will not significantly affect our results. For the collected library migrations, we do not know how complete or representative they are for all migrations in the wild. However, projects with good commit messages about library migration may be of better quality than those without, so their practices should be more valuable for reference by other practitioners. The manual identification of migrations, the resolution of application domains, and the thematic analysis process may all contain labeling errors. To mitigate this threat, at least two authors double check the results for the first two steps and conduct independent coding for the thematic analysis.

7.2.2 External Validity. Our findings may not generalize to migrations that happen without being mentioned in commit messages and migrations in other projects and between other libraries. We mitigate this threat by collecting a large number of starred GitHub projects and library migrations for popular libraries, so that our results will be useful for a broad audience of Java practitioners. Our results may also not generalize to proprietary projects and projects in other programming languages. Intuitively, the results may differ because of difference in application domain and stakeholder interest, so we encourage future work to investigate this direction.

8 CONCLUSION AND FUTURE WORK

In this paper, we conduct a descriptive mixed methods study on *how* and *why* library migrations happen in Java software projects. Our contributions are 1) A framework for quantifying dependency changes and library migrations based on version control data. 2) Findings regarding prevalence, trends and rationales of library migration in Java projects. 3) A largest dataset to-date in the topic of library migration. As future work, we plan to understand the decision-making process of library migration, and evaluate the practicality of existing automated approaches for library migration.

[24] https://github.com/openzipkin-attic/zipkin-sparkstreaming/issues/30

REFERENCES

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2021. Empirical analysis of security vulnerabilities in Python packages. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[2] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. How does library migration impact software quality and comprehension? An empirical study. In *International Conference on Software and Software Reuse*. Springer, 245–260.

[3] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. 2020. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing* 90 (2020), 106140.

[4] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 347–357.

[5] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. 2010. Swing to SWT and back: Patterns for API migration by wrapping. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[6] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs Van Der Storm. 2009. Study of an API migration for two XML APIs. In *International Conference on Software Language Engineering*. Springer, 42–61.

[7] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (2015), 1275–1317.

[8] Cédric Beust. 2019. TestNG - Welcome. https://testng.org/

[9] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. 2009. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 1–10.

[10] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008.

[11] Virginia Braun and Victoria Clarke. 2012. Thematic analysis. (2012).

[12] Chunyang Chen and Zhenchang Xing. 2016. Similartech: automatically recommend analogical libraries across different programming languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 834–839.

[13] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. 2019. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* (2019).

[14] Jailton Coelho and Marco Tulio Valente. 2017. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 186–196.

[15] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E Hassan. 2019. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering* (2019).

[16] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E Hassan. 2021. Deprecation of packages and releases in software ecosystems: A case study on npm. *IEEE Transactions on Software Engineering* 01 (2021), 1–1.

[17] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael FP O'Boyle. 2020. M3: Semantic API Migrations. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 90–102.

[18] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 275–284.

[19] Bradley E Cossette and Robert J Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[20] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. 2015. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 109–118.

[21] Russ Cox. 2019. Surviving software dependencies. *Commun. ACM* 62, 9 (2019), 36–43.

[22] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 275–284.

[23] Fernando López de la Mora and Sarah Nadi. 2018. An empirical study of metric-based comparisons of software libraries. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. 22–31.

[24] Fernando López De La Mora and Sarah Nadi. 2018. Which library should I use?: a metric-based comparison of software libraries. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 37–40.

[25] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 404–414.

[26] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 181–191.

[27] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[28] Vaclav Dedik. 2015. Replace org.json with jackson. https://github.com/release-engineering/pom-manipulation-ext/commit/d7d4a0305e6b681e97e79b0510ca0b7570f2ed00

[29] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 349–359.

[30] FasterXML, LLC. 2019. FasterXML/jackson: Main Portal page for the Jackson Project. https://github.com/FasterXML/jackson

[31] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. 2007. Model-driven engineering for software migration in a large industrial context. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 482–497.

[32] Python Software Foundation. 2021. PyPI: the Python package index. https://pypi.org/

[33] The Apache Software Foundation. 2021. Apache Maven Project. https://maven.apache.org/

[34] GitHub, Inc. 2021. Autolinked references and URLs. https://docs.github.com/en/github/writing-on-github/autolinked-references-and-urls

[35] GitHub, Inc. 2021. GitHub Advisory Database. https://github.com/advisories

[36] Google, Inc. 2019. google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson

[37] Hao He, Yulin Xu, Yixiao Ma, Yifei Xu, Guangtai Liang, and Minghui Zhou. 2021. A multi-metric manking approach for library migration recommendations. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[38] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E Hassan. 2016. Logging library migrations: A case study for the apache software foundation projects. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 154–164.

[39] Jeremy Katz. 2020. *Libraries.io Open Source Repository and Dependency Metadata*. https://doi.org/10.5281/zenodo.3626071

[40] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. 2019. Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 476–487.

[41] Klaus Krippendorff. 2011. Computing Krippendorff's alpha-reliability. (2011).

[42] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.

[43] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, and Katsuro Inoue. 2015. Trusting a library: A study of the latency to adopt the latest maven release. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 520–524.

[44] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

[45] Hemank Lamba, Asher Trockman, Daniel Armanios, Christian Kästner, Heather Miller, and Bogdan Vasilescu. 2020. Heard it through the Gitvine: an empirical study of tool diffusion across the npm ecosystem. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 505–517.

[46] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting Android API migrations using code examples. *IEEE Transactions on Software Engineering* (2020).

[47] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 245–256.

[48] Snyk Limited. 2021. Snyk | Developer security | Develop fast. Stay secure. https://snyk.io/

[49] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of code: an infrastructure for mining the universe of open source VCS data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 143–154.

[50] Yuxing Ma, Audris Mockus, Russell Zaretzki, Bogdan Bichescu, and Randy Bradley. 2020. A Methodology for Analyzing Uptake of SoftwareTechnologies Among Developers. *IEEE Transactions on Software Engineering* (2020).

[51] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd*

IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 84–94.
[52] Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering* 12, 5 (2007), 471–516.
[53] MvnRepository. 2021. Maven Central Repository. https://mvnrepository.com/repos/central
[54] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 651–654.
[55] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. CrossRec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software* 161 (2020), 110460.
[56] npm, Inc. 2021. npm | Build amazing things. https://www.npmjs.com/
[57] Ola Johansson and Markus Olausson. 2021. AlternativeTo: Crowd-Sourced Software Recommendation. https://alternativeto.net/
[58] Oracle. 2021. JAR File Specification. https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html
[59] Amantia Pano, Daniel Graziotin, and Pekka Abrahamsson. 2018. Factors and actors leading to the adoption of a JavaScript framework. *Empirical Software Engineering* 23, 6 (2018), 3503–3534.
[60] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
[61] Rebecca Passonneau. 2006. Measuring agreement on set-valued items (MASI) for semantic and pragmatic annotation. (2006).
[62] QOS.ch. 2019. Logback Project. http://logback.qos.ch/
[63] QOS.ch. 2019. SLF4J: Simple Logging Facade for Java. http://www.slf4j.org/
[64] Patrick Riehmann, Manfred Hanfler, and Bernd Froehlich. 2005. Interactive sankey diagrams. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* IEEE, 233–240.
[65] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. 2018. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software* 145 (2018), 164–179.
[66] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? Confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 858–870.
[67] WhiteSource Software. 2021. WhiteSource: Open Source Security and License Management Solution. https://www.whitesourcesoftware.com/
[68] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The emergence of software diversity in maven central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 333–343.
[69] StackOverflow Users. 2020. Can I squash a merge commit with one of its predecessors? https://stackoverflow.com/questions/61906251/can-i-squash-a-merge-commit-with-one-of-its-predecessors
[70] Xin Tan and Minghui Zhou. 2019. How to communicate when submitting patches: An empirical study of the linux kernel. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–26.
[71] Cedric Teyton, Jean-Remy Falleri, and Xavier Blanc. 2012. Mining library migration graphs. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 289–298.
[72] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2013. Automatic discovery of function mappings between similar libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 192–201.
[73] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A study of library migrations in Java. *Journal of Software: Evolution and Process* 26, 11 (2014), 1030–1052.
[74] The Apache Software Foundation. 2012. Apache Log4j 1.2. https://logging.apache.org/log4j/1.2/
[75] The Apache Software Foundation. 2014. Apache Commons Logging - Overview. https://commons.apache.org/proper/commons-logging/
[76] The Apache Software Foundation. 2020. Log4j - Apache Log4j 2. https://logging.apache.org/log4j/2.x/
[77] The JUnit Team. 2020. JUnit - About. https://junit.org/junit4/
[78] Ferdian Thung, David Lo, and Julia Lawall. 2013. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 182–191.
[79] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 644–655.
[80] Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Laurent Deruelle, Stéphane Ducasse, and Mustapha Derras. 2019. GUI migration using MDE from GWT to angular 6: an industrial case. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 579–583.
[81] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.
[82] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. 2020. Why reinventing the wheels? An empirical study on library reuse and re-implementation. *Empirical Software Engineering* 25, 1 (2020), 755–789.
[83] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 335–346.
[84] Likang Yin and Vladimir Filkov. 2020. Team discussions and dynamics during DevOps tool adoptions in OSS projects. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 697–708.
[85] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*. Springer, 95–110.
[86] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 195–204.
[87] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.