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

Hao He
Department of Computer Science and Technology, Peking University, and Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing, China
heh@pku.edu.cn

Runzhi He
Department of Computer Science and Technology, Peking University, and Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing, China
rzhe@pku.edu.cn

Haiqiao Gu∗
Department of Physics, Tsinghua University
Beijing, China
ghq17@mails.tsinghua.edu.cn

Minghui Zhou†
Department of Computer Science and Technology, Peking University, and Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing, China
zhmh@pku.edu.cn

ABSTRACT

With the rise of open-source software and package hosting platforms, reusing 3rd-party libraries has become a common practice. Due to various failures during software evolution, a project may 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 (41.04%) projects having at least one library removal, 1,564 (7.96%, lower-bound) to 5,004 (25.46%, upper-bound) projects have at least one migration, and a median project with migrations has 2 to 4 migrations in total. We discover that library migrations are dominated by several domains (logging, JSON, testing and web service) 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 (e.g., lack of maintenance, usability, integration, etc), 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. We provide a replication package at https://doi.org/10.5281/zenodo.4816752.

CCS CONCEPTS

• Software and its engineering → Software libraries and repositories; Maintaining software; Software evolution.

KEYWORDS

library migration, mining software repositories, evolution and maintenance, empirical software engineering

ACM Reference Format:

1 INTRODUCTION

Reusing existing 3rd-party libraries† with ready-to-use features has long been a common practice in software development, which can increase software quality and development productivity [49]. Still, it is not until the rise of open-source software and the availability of central hosting platforms (e.g. Maven [50], NPM [52], PyPI [29], 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 [50]. 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 or even hundreds of existing libraries [76, 80].

Meanwhile, the adoption of 3rd-party libraries brings unique challenges in the entire software life-cycle [20]. 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

†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 [54]) available for download in a central hosting platform (e.g. Maven Central [50]). 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.

Works done at Peking University
Corresponding Author

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2021 Copyright held by the owner/author(s).
https://doi.org/10.1145/3468264.3468571
complex socio-technical factors need to be considered [55, 78]. Second, increasing concerns are being raised about the risk of using outdated libraries as they may contain security vulnerabilities and unresolved issues [1, 24, 56, 86], 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, 43, 84] and practitioners propose industry solutions (e.g., Synk [45], WhiteSource [64], and GitHub Advisories [32]) 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 [35–37, 69, 71].

While a large body of existing researches are conducted to characterize and understand library adoption [22, 39, 44, 47, 55, 78, 81, 83], and library updates [7, 15, 19, 23, 27, 42, 43, 48, 65, 84, 86], researches on library migrations [2, 5, 6, 37, 69, 71] 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 when developers 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, while existing studies [69, 71] suffer from methodological limitations (details in Section 2). To address these challenges, we propose an event-based dependency change model, formulate library migration based on dependency changes and self-admitted commit messages, retrieve migrations using a state-of-the-art mining algorithm [36], 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, and apply thematic analysis [11, 21] 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.

4. 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.

We provide a full replication package at Zenodo. The scripts and documentation can also be accessed in this GitHub repository.

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 [43], one API to another API [4], one programming language to another programming language [85], one platform to another platform [79], or one library to another library [36, 37, 69, 71]. In this paper, we use the term library migration to refer to the process of replacing one library with another 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 $(l_1, l_2)$ 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 [37]. Such discussions may not result in a migration if no consensus is reached or the perceived benefits do not outweigh the costs [37]. 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 aim to improve development efficiency for the third step, by using API wrappers [5, 6], mining API mappings [3, 4, 13, 70], or directly editing code to use the new API [17, 82]. Studies also show that library migrations may improve code quality [2], but performance is rarely improved [37].

In this paper, we focus on understanding the first two steps of library migration, following existing studies [37, 69, 71] (Table 1).

---

2. https://github.com/hehao98/LibraryMigration
3. More precisely, by another we mean the names (i.e., groupId:artifactId in Java) of the two libraries are different.
Teyton et al. [69] 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 [71], 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. [37] 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 existing studies [37, 69, 71] have several limitations. In particular, [69] only obtain a small number of migrations; [69, 71] do not have a solid methodology foundation and are hard to reproduce. By their definition, a library migration happens if two libraries with similar functionalities co-exist and one gets removed in project history. The notion of “similar” is ambiguous and “library migrations” by this definition may be over-estimations. They also do not provide sufficient details about their manual identification criteria, nor do they share their dataset of library migrations. Furthermore, [71] uses only source code analysis for identifying libraries, which may miss migrations that only modify configuration files (e.g., migrate logger⁶ or database⁷). Finally, [37] focuses on a relatively small number (49) of logging library migrations. Consequently, the findings in [37] cannot reflect general trends and may not generalize to other kind of libraries. In this study, we mine large-scale historical data 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 [22, 39, 44, 47, 55, 78, 81, 83] and library updates [7, 15, 19, 23, 27, 42, 43, 48, 65, 84, 86]. In terms of library adoption, researchers use interviews and surveys to summarize developer considerations when choosing JavaScript frameworks [55], deciding whether to re-implement [81], and selecting libraries in general [78]. Researchers also employ different modeling approaches to identify factors that lead to adoption between R dataframe libraries [47] and between JavaScript CI/CD tools [39, 44, 83]. In terms of library update, researchers have investigated how developers update [7] or not update libraries [19, 23, 42, 43, 65, 84], mechanisms to support library update [27, 48], and the impact of security vulnerabilities introduced by outdated libraries [24, 86]. 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.

1https://github.com/eclipse/leshan/commit/d103207
2https://github.com/mybatis/migrations-maven-plugin/commit/1b1c609

### 3 DATA COLLECTION

#### 3.1 Collecting Projects and Libraries

We begin with the latest Libraries.io dataset [38] (released in January 2020), which is widely used in related research (e.g. [1, 25, 36, 84]). 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 [37, 69, 71]. 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-the-art practice by mature projects. Then, to retrieve version control data for these projects, we use the World of Code database [46] (version R, constructed in April 2020). To simplify the task of dependency extraction, we focus on projects that use Maven [30] 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 [50], and 2) have been included in the pom.xml of more than 10 repositories in the aforementioned 19,652 repositories. Finally, we get 4,022 libraries.⁸ 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.⁹

---

1Following 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.
2The only exception is io.prometheus:simpleclient_common which describes itself as common code used by various modules of the simpleclient.

---

### Table 1: Summary of Related Work on Library Migration

<table>
<thead>
<tr>
<th>Paper</th>
<th>Subject of Study</th>
<th>Reported Prevalence</th>
<th>Reported Reasons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teyton et al. [69]</td>
<td>80 migration rules</td>
<td>N.A.</td>
<td>convenience, outdated, incompatibilities</td>
</tr>
<tr>
<td>Teyton et al. [71]</td>
<td>329 migration rules, 1,198 migrations, 26 commit messages</td>
<td>5.57% of 15,168 projects</td>
<td>feature, performance, configuration, bug, environment</td>
</tr>
<tr>
<td>Kabinna et al. [37]</td>
<td>49 logging library migration attempts (33 successful)</td>
<td>33 of 223 ASF projects (14.80%)</td>
<td>feature, performance, flexibility, reduce future maintenance, reduce dependencies</td>
</tr>
</tbody>
</table>
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. Our approach is to first extract dependency changes, then identify commits between library migrations, and finally, create migration rules based on these identified commits.

Dependency Changes: We define a dependency change as a difference in the dependencies of a library before and after a commit. The changes can be categorized into three types: adoptions ($D^+$), removals ($D^-$), and versions ($D^v$). The set of adoption events is $D^+ = \{ (l, c) | \forall v \in D^c, \exists v' \in D^l, \text{ and } v' = v \}$, where $l$ is a library, and $c$ is a commit that adds a new dependency. The set of removal events is $D^- = \{ (l, c) | \forall v \in D^c, \exists v' \in D^l, \text{ and } v' = v \}$, where $v$ is a dependency. The set of version events is $D^v = \{ (l, c) | \forall v \in D^c, \exists v' \in D^l, \text{ and } v' = v \}$, where $v$ and $v'$ are versions of the same dependency.

Commit Identification: To identify commits related to library migrations, we first extract all commits that modify pom.xml files. Then, we compute the dependency changes based on the version control events in pom.xml files. We use a state-of-art mining algorithm to identify migration commits from these dependency changes.

Migration Rules: We then create migration rules based on the identified migration commits. A migration rule is defined as a set of adoption events, removal events, and version events. We use a combination of human inspection and automatic rule generation to create these migration rules. The rules are then validated by human reviewers.

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. We need to determine whether a migration is feasible between two arbitrary library pairs (i.e., for $l_1$ and $l_2$, whether $(l_1, l_2)$ is a migration rule). The challenge is 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 [26]), and define $(l_1, l_2)$ as a migration rule if and only if we can find a migration commit that conducted the migration from $l_1$ to $l_2$.

We use a combination of human inspection and automatic rule generation to create these migration rules. The rules are then validated by human reviewers. For each source library query, the algorithm mines possible target libraries, ranks 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 [69], 2) 500 most popular libraries by number of adopters 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 the results, resulting in 393 rename rules. Teyton et al. [71] exclude rename rules by manually mapping groupId:artifactId to “library names.” However, library names are 1) very common at least in Java/Maven, 2) developers also state them as migrations in commit messages, and 3) their APIs are often significantly reworked. Thus, we choose to retain rename rules in subsequent analysis as a special type of migration rule, and distinguish them from other rules where their property significantly differs.

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

### Table 2: Statistics of the Collected Dataset

<table>
<thead>
<tr>
<th>Data</th>
<th>Size/Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects ($P$)</td>
<td>19,652</td>
</tr>
<tr>
<td>Stars</td>
<td>median = 32, stddev = 1407.93</td>
</tr>
<tr>
<td>Commits</td>
<td>median = 142, stddev = 5614.70</td>
</tr>
<tr>
<td>Active Months</td>
<td>median = 27, stddev = 26.97</td>
</tr>
<tr>
<td>pom.xml Files</td>
<td>median = 2, stddev = 75.72</td>
</tr>
<tr>
<td>Libraries ($L$)</td>
<td>4,022</td>
</tr>
<tr>
<td>Versions</td>
<td>median = 26, stddev = 104.20</td>
</tr>
<tr>
<td>Included Repositories</td>
<td>median = 25, stddev = 310.94</td>
</tr>
<tr>
<td>Dependency Changes ($D$)</td>
<td>2,629,992 (302,774 commits)</td>
</tr>
<tr>
<td>Adoptions ($D^+$)</td>
<td>1,771,664 (208,691 commits)</td>
</tr>
<tr>
<td>Removals ($D^-$)</td>
<td>290,150 (56,377 commits)</td>
</tr>
<tr>
<td>Version Changes ($D^v$)</td>
<td>568,178 (97,447 commits)</td>
</tr>
<tr>
<td>Migration Rules ($R$)</td>
<td>1,194 (393 rename)</td>
</tr>
<tr>
<td>Migration Commits ($M$)</td>
<td>3,163 (1,459 rename)</td>
</tr>
</tbody>
</table>
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., $P_r = \{p \mid p \in \mathcal{P} \land D_p \neq \emptyset\}$, 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 $D \in \{D_c, D^r_c, D^v_c, D^a_c\}$), and 2) the average number of commits between changes for all change types (i.e., $|C_p|/|\{c \in C_p \land D \neq \emptyset\}|$ for $D \in \{D_c, D^r_c, D^v_c, D^a_c\}$). We use the latter to estimate the frequency (or interval) of each change type along the project development history.

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, 8,657 (47.43%) have at least one version change and 8,045 (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 $\rho$ 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 ($\rho = 0.158$, $p < 0.001$).

4.2 RQ1.2: Migration Frequency Analysis

4.2.1 Methodology. We begin with $M$ and $R$ 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 $P_m = \{p \mid p \in \mathcal{P} \land \exists l \in L_m \land l \in D_p^r\}$. The computation of frequency for library migrations is difficult because we cannot guarantee the completeness of mined migrations for any non-trivial mining approach [36, 69]. As mentioned in Section 3.3, $M$ and $R$ 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 $\langle l_1, l_2 \rangle \in R \land l_1 \in D_{p,f}^r \land l_2 \in D_{p,f}^v$, 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
we can observe a clear increasing trend (note that we only have partial data for 2020). This indicates that library migrations are becoming increasingly prevalent in the Java/Maven ecosystem since 2006.

Finally, for each migration commit \( m \in M \), we use its commit timestamp to plot a longitudinal trend of library migrations.

4.2.2 Results. Among the 17,426 projects (\( P_{m} \)), 7,950 (45.62%) have at least one removal of libraries in \( h_{m} \), 3,004 (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. 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 \( P_{m} \)), 2,161 (56.93%) / 764 (20.13%) projects have at least one possible / confirmed migration. Furthermore, 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, and we can observe a clear increasing trend (note that we only have partial data for 2020). This indicates that library migrations are becoming increasingly prevalent in the Java/Maven ecosystem since the publication of Teyton et al. [71].

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. [69, 71] as the basis to facilitate further analysis. We define the migration graph in our study as a weighted directed graph \( \langle L_{m}, R, w \rangle \), where libraries in \( L_{m} \) are nodes, migration rules in \( R \) are edges, and \( w() \) is an edge weighting function. For \( (l_{1}, l_{2}) \in R \), 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 \( M \).

To demonstrate the extend 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 \( l \in L_{m} \):

\[
flow(l) = \frac{\text{deg}^{-}(l) - \text{deg}^{+}(l)}{\text{deg}^{-}(l) + \text{deg}^{+}(l)}
\]

where \( \text{deg}^{-}(l) \) is the weighted indegree for node \( l \) and \( \text{deg}^{+}(l) \) 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 7.

Figure 7: Distribution of \( \mathcal{K} \) by application domain.

Figure 8: Number of migrations for migration rules in each application domain (\( \sum_{p \in \mathcal{P}} \text{MigCnt}(p) \), Equation 1).

is is unidirectional in that most libraries are either mostly adopted or mostly abandoned.

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.

Figure 9: Distribution of \( \text{flow}(l) \) under different conditions.

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 10. We exclude the sub-graph for web service due to space constraints. To make the graphs readable, we only retain migration rules \( (l_1, l_2) \) with \( w(l_1, l_2) \geq 10 \) for testing and JSON, and \( w(l_1, l_2) \geq 15 \) for logging. The unidirectional trend is clearly visible using Sankey diagrams. In the case of logging libraries, commons-logging [73] and log4j [74] are two mostly abandoned libraries, while libraries under group ID org.apache.logging.log4j (i.e., log4j 2 [74]) are mostly adopted. Libraries related to slf4j [59] and logback [58] are somewhere in between. The reason for such phenomenon may be that the Java community has two major logging practice changes [37]: 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 [33] is first adopted but then switched to jackson [28]), 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 10 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 [75] and org.testng:testng [8], which may be the results of early competition when both libraries are immature.
Figure 10: 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.

6 RQ3: WHAT ARE THE FREQUENTLY MENTIONED REASONS WHEN DEVELOPERS 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 when developers 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 ($M$, Section 3.3). To extend available data, we follow GitHub conventions to identify issue references in commit messages [31], and retrieve pull requests containing the commits using the GitHub RESTful APIs. 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, 21], which is frequently used to extract recurring patterns from text in software engineering research (e.g., [14, 62, 68]).

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 [37, 71] (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 [57] as the distance measure and Krippendorff’s alpha [40] 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 [41]. 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. During the conflict resolution process, the two aforementioned authors first discuss about the relevant text content to see if anyone has misunderstandings or labeling mistakes. If the text contains insufficient information or ambiguities, they may search over the Web, inspect the project development history, and infer the underlying reasons based on their prior experience. One author has 4 years of Java development experience and arbitrates most of the cases. Cases where the conflicts cannot be resolved will be simply labeled 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...
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 §).

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

(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 because of advantages in the target library. Inspired by Larios Vargas et al. [78], we divide them into the following 7 dimensions.

- Usability. The most frequently mentioned reason in target library (Table 1), and 5 reasons (marked with §) originate from existing work but are further extended and re-organized based on our findings.

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 1og4j to 1og4j 2, s1f4j or logback, e.g. Fix for log4j vulnerability CVE-2019-17571. 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. [78], 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). 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.

Some projects also mention that choosing an easy-to-use library is important for maintainability in the future. Kabir et al. [37] mention “flexibility” and Teyton et al. [69] 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. The mentioned features are highly diverse across different application domains and project contexts.

Performance. Performance is also frequently mentioned as a reason for migration. It may be because it is too trivial to be mentioned or discussed, or because the library has unfixed issues and security vulnerabilities. However, since our findings reveal that library migration is 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 more publicly visible. For example, package hosting platforms can compute and demonstrate the current retention rate for a library, as more publicly visible. For example, package hosting platforms can compute and demonstrate the current retention rate for a library, as more publicly visible. For example, package hosting platforms can compute and demonstrate the current retention rate for a library.

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.

Popularity. Surprisingly, popularity is not a frequently mentioned reason, and most of the cases are because of library renames (41.71%). It may be because it is too trivial to be mentioned or discussed, or because the library’s name 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.

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.

6.2.3 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 the 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. 2) To avoid conflicts/incompatibility with other dependencies or project modules, e.g., Removes the PowerMock dependency which was having some bad interactions with Mockito. 3) To be compatible with the required runtime environment, e.g., migrate from lombok to kotlin...to a fix a bug with java 11. Tayton et al. [71] 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. 2) Make Logging Frameworks Coherent Across Libraries and IOT Codebase. 3) Lots of cleanup. Removed Junit from a couple of modules (everything is TestNG now).

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, refrain from using org.json:json because of its unusual license term The Software shall be used for Good, not Evil. 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 JLLine2 as a dependency since it’s not approved by LocationTech.

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.

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. AlternativeTo [53] 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 14% of library migrations happen because of unmaintained libraries, and the situation significantly worsens if an unmaintained library has unresolved issues and security vulnerabilities. However, open-source libraries are prone to sustainability failures, and unmaintained libraries may have vulnerabilities which may be even unknown to public but already discovered by malicious parties.

---

15https://github.com/apache/incubator-pinot/pull/2473
16https://github.com/apache/incubator-pinot/pull/439
17Commit ce45e04 in https://github.com/azagniotov/stubby4j
18https://github.com/sarl/sarl/issues/875
19Commit 9b5e384 in https://github.com/eclipse/hono/
20Commit 2aad070 in https://github.com/GoogleCloudPlatform/java-docs-samples/
21https://github.com/spotify/ffwd/pull/140
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 pom.xml 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. [78], only a few of them serve as the major driving force 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, maven 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,27 but it is not obvious from Spark documentation. Therefore, we also suggest library developers, especially framework developers, should explicitly document what libraries are recommended to use together with their library and what libraries are not recommended. Researchers may also consider developing automated tools to alleviate this issue, e.g., by applying existing library recommendation approaches [51, 61, 76] to discover co-usage patterns, or design new approaches to detect common integration pitfalls.

7.2 Threats to Validity

7.2.1 Internal Validity. Extracting dependencies from pom.xml is the standard practice by many recent studies (e.g., [19, 27, 43, 80]), but it comes with a number of limitations. First, the declared dependencies may not be actually used (i.e., bloated) and used dependencies may be transitively included and not explicitly declared [66]. Second, in multi-module projects, dependencies may be moved between different modules, which may cause over-estimations on removals and adoptions. Our approach for computing dependency changes may also generate errors in some corner cases (e.g., a merge commit contains conflict resolution or is squashed with its parents [67]). All these cases will induce noises into our collected dependency changes and affect the results in RQ1, but they are generally recognized as bad practice and should not be very common in well maintained projects. Additionally, we use self-admitted commit messages as the final criteria to ensure that all migrations we use to answer RQ2 and RQ3 are real migrations. 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. Our findings may also not generalize to projects that do not use build automation tools or projects that use other build automation tools (e.g., Graddle [34]), but Maven is the most popular tool [63] and the choice of automation tools should not significantly affect project properties. Therefore, we believe our results will be useful for a broad audience of Java practitioners. Finally, our results may not generalize to proprietary projects and projects in other programming languages / ecosystems (e.g., JavaScript/npm, Python/PyPI, etc). Intuitively, the results may differ because of difference in application domain, developer preferences, and stakeholder interest, and we leave the investigation of ecosystem differences as future work.

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.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China Grant 2018YFB1004201 and the National Natural Science Foundation of China Grant 61825201. We would also like to thank the anonymous reviewers for their constructive feedback.