Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study

Haiqiao Gu, Hao He, Minghui Zhou*
School of Computer Science, Peking University, China
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
ghq@stu.pku.edu.cn, {heh, zhmh}@pku.edu.cn

***Abstract*—Reusing open-source software libraries has become the norm in modern software development, but libraries can fail due to various reasons, e.g., security vulnerabilities, lacking features, and end of maintenance. In some cases, developers need to replace a library with another competent library with similar functionalities, i.e., *library migration*. Previous studies have leveraged library migrations as a unique lens of observation to reveal insights into library selection and dependency management in general. However, they are heavily biased toward Java while the generalizability of their findings remains unknown.**

In this paper, we present a comparative study on self-admitted library migrations (SALMs) from three packaging ecosystems: Java/Maven, JavaScript/npm, and Python/PyPI. For this study, we design a set of semi-automatic methods that accurately locate SALMs, their domains, and their rationales from git repositories. We reveal that SALMs are prevalent and highly unidirectional in all three ecosystems, and the underlying rationales can be well covered by a previous theoretical framework. Also, SALMs in these ecosystems present domain similarity (testing frameworks, web frameworks, HTTP clients, and serialization). However, we observe differences in the longitudinal trends, the distributions of rationales, the ecosystem-specific domains, and the levels of unidirectionality, all of which indicate that Python/PyPI sees increasingly intense competition between libraries and deserves more research on library recommendation and migration.

***Index Terms*—software packaging ecosystem, library migration, mining software repositories, cross-ecosystem comparison**

I. Introduction

The reuse of open-source software libraries[1] has long been a common practice in software development which brings various quality, productivity, and economic benefits [5]. In the recent decade, the number of publicly available libraries has soared in major software packaging ecosystems [6] (e.g., Java/Maven [7], JavaScript/npm [8], Python/PyPI [9]) and complex dependency networks have been formed in each of them [10]. Software projects belonging to, or reusing libraries from such ecosystems, can establish dependencies on a large number of libraries in that ecosystem [11], [12].

The growing number of libraries and dependencies means that software projects nowadays, both open-source and proprietary, need to pay more and more attention to the selection and management of their dependent libraries (i.e., dependency management) [13]. Typically, developers need to select appropriate libraries for their project from a large choice space [14], manage the versions of each library [15], [16], and react to critical library failures (e.g., lacking features or performance [4], end of maintenance [17], security vulnerabilities [18]). Such critical library failures can be sometimes resolved by an update or a workaround, but often need to be resolved by replacing the library with another competent library offering the same or similar functionalities [4], [13]. This practice is referred to as *library migration* in the existing literature [1]–[4].

Researchers have devoted substantial effort to the investigation of library migration (e.g., popularity, domains, rationales, and directionality). In particular, researchers find that library migrations 1) are prevalent in Java/Maven [1]–[4], 2) tend to be unidirectional [4], and 3) are driven by a number of socio-technical factors from the source library, the target library, and project integration [1], [2], [4]. Their study subjects are all Java/Maven, which has a long history and many ecosystem-specific peculiarities. However, library migrations in other packaging ecosystems may demonstrate different patterns and it remains unknown to what extent the findings on Java/Maven can be generalized to other ecosystems.

Therefore, we aim to verify the generalizability of results and implications from previous library migration studies in a broader context. Toward this goal, a comparative analysis of major packaging ecosystems is necessary. In this study, we target the following ecosystems: Java/Maven, JavaScript/npm, and Python/PyPI, as they rank top-3 in terms of ecosystem size [6], their programming languages rank top-3 in GitHub in terms of popularity [19], and they cover diverse application domains (e.g., mobile, web, AI). We believe an analysis and comparison of the three ecosystems can yield results and implications applicable to a broad developer audience.

Identifying library migrations from software repository data is non-trivial: previous approaches either suffer from inaccuracies or require a huge amount of manual effort [20], [21]. To enable large-scale analyses and cross-ecosystem comparisons, we study *self-admitted library migrations* (SALMs), i.e., library migrations explicitly admitted by developers in the commit messages. We seek to empirically investigate their

*Minghui Zhou is the corresponding author.

[1]Developers may use several different terms (e.g., library, package, framework, component) to refer to a piece of reusable software. This paper uses the term *library* to refer to all of them for consistency with prior work [1]–[4].

prevalence, domains, rationales, and directionality, to explain *how* and *why* SALMs occur and compare among ecosystems. More specifically, we ask the following research questions:

- **RQ1:** How common are SALMs in Java/Maven, JavaScript/npm, and Python/PyPI? How do the longitudinal trends differ in the three ecosystems?
- **RQ2:** In what library domains do SALMs happen? How do the domains differ in the three ecosystems?
- **RQ3:** What are the rationales for SALMs? How do the rationales differ in the three ecosystems?
- **RQ4:** Are SALMs unidirectional? How does the directionality differ in the three ecosystems?

To answer the **RQ**s, we design an NLP-powered heuristic-based mining algorithm to automatically mine SALMs. By applying the algorithm to 177,378 GitHub repositories using libraries from Java/Maven, JavaScript/npm, or Python/PyPI, we obtain 33,667 SALMs with 84.70%, 83.80%, and 86.50% precision, respectively. We further apply a multitude of automated and manual methods to this dataset to answer the **RQ**s.

Our study reveals a set of consistencies and discrepancies regarding how and why SALMs happen in the three packaging ecosystems. Although many previous findings about library migration still hold in all of them, there are differences in terms of longitudinal trends, application domains, rationales, and directionality. Notably, the rising number of migrations and the lower directionality in Python/PyPI indicate that Python/PyPI is an ecosystem with many competitor libraries and more research effort should be directed to help Python developers select and migrate their libraries.

II. Related Work

Nowadays, software projects are built upon a large number of libraries (i.e., dependencies) and developers believe such reuse leads to higher productivity, increased code quality, and lower development costs [5]. However, this high level of reuse also brings developers the new and increasingly important task of *dependency management* in which many new problems need to be addressed [13]. In this section, we review related work in three key stages of dependency management: library selection, library version management, and library migration.

A. Library Selection

For any highly-demanded feature or problem domain, it is likely that several competitor libraries exist and developers need to choose the most appropriate one for their projects [22]. Kavaler et al. [23] find that the adoption of different JavaScript quality assurance tools (available in npm as "packages") can lead to different software maintenance outcomes. To identify factors driving library selection, researchers have investigated several cases in depth (e.g., trivial packages [24], JavaScript frameworks [25], DevOps tools [26], R data tables [27]); they find that library selection can be driven by a multitude of technical, social, and economic factors (e.g., performance, ease of use, community atmosphere, social network influence). Similar factors are also obtained from a more general industrial interview study by Vargas et al. [14]. Given the multitude of factors, selecting non-trivial libraries can be a demanding task requiring rich development experience and even a prototyping trial [14], [25], [26]. To assist library selection, researchers propose solutions to mine the differences of similar libraries from Stack Overflow [28] or to aggregate important library metrics in an IDE plugin [29].

B. Library Version Management

After library adoption, developers need to spend additional development effort to maintain the versions of their libraries. It is generally suggested as best practice that libraries should be kept up-to-date to ensure project maintainability and supply chain security in the long term [30], [31]. However, keeping libraries up-to-date brings extra development costs due to the risks of breaking changes [32] and many software projects still use outdated dependencies [15]. Bavota et al. [33] find that library updates in the Apache ecosystem are driven by major new features but hindered by API removals. To make library updates easier, researchers and practitioners have proposed and evaluated various automated solutions, such as dependency management bots [34]–[36] and API adaptation tools [37], [38]. For projects with a large number of dependencies, developers may also need to handle version-related issues, such as dependency conflicts [39], [40], version inconsistencies [41], and incorrect version specifications [42], [43].

C. Library Migration

A software project and its dependencies generally evolve in an asynchronous manner and project developers often have little control over the development of their dependencies. Thus, it is common for a library (denoted as l_1) to have unexpected failures or inability to meet the project's expectations, even if they are kept up-to-date. In such cases, developers need to replace the library l_1 with another library l_2 with similar functionalities. We consider this process as a *library migration*. In some sense, library migrations can be viewed as the direct consequence of improper library selections and lessons can be learned in hindsight. Following previous work [1], [4], 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 remainder of this paper.

Library migrations are hard to study because it is challenging to locate ground truth migrations in the wild. Previous studies either combine heuristic-based algorithms with manual validation to mine library migrations from git repositories [1], [3], [4], [20], [44], [45], or search for relevant discussions in the issue tracker [2]. They report from their analyses that:

- Library migrations are prevalent in Java/Maven and more likely to happen in mature projects with a large number of dependencies [1]–[4], [20];
- Library migrations tend to be unidirectional [4], [44];
- Library migrations are driven by a number of socio-technical factors from the source library, the target library, and project integration [1], [2], [4], [44];
- Library migrations can improve code quality and reduce the complexity of implementation [3], [44], but performance is rarely improved for logging libraries [2];

- Library migrations are error prone and difficult [2], [44].

There is also research effort on the automation of library migrations via wrapping APIs [46], [47] or mining API mappings [48]–[50], and on the recommendation of library migrations [21], [51]. However, almost all previous studies are based on the Java/Maven ecosystem with only two exceptions [44], [45]: the former [44] is a case study on two Python testing frameworks while the latter [45] builds a Python library migration dataset. Neither of which contains large-scale empirical analysis on multiple ecosystems and application domains. The bias toward Java/Maven in the library migration literature may hinder the application of their findings to developer communities using different programming languages & packaging ecosystems, similar to the API evolution literature [52].

In this study, we seek to investigate to what extent are the findings and implications of previous studies on library migration generalizable to a broader context (i.e., three packaging ecosystems, no specific domain). We expect such a study to offer a unique perspective and insights into library selection, library migration, and dependency management in general.

III. Methodology

A. Subjects of Study

Following prior studies [4], [10], [12], we consider Maven, npm, or PyPI as packaging ecosystems and the versioned artifacts hosted on Maven, npm, or PyPI as *libraries*. We consider GitHub repositories that depend on libraries from Maven, npm, or PyPI as *projects*. We focus on library migrations that happened in the development history of projects depending on libraries from one of the packaging ecosystems. However, the identification of library migrations from project development histories can be challenging: manual inspection would require a prohibitive amount of effort, while the soundness of automated mining approaches is hard to guarantee [4]. Our insight is to study *self-admitted library migrations* (SALMs), i.e., library migrations explicitly documented in commit messages. Note that a GitHub repository can also host development for a library, so our study covers migrations in the development of both libraries and downstream applications.

1) Libraries: We utilize the Libraries.io dataset [53] (version 1.6.0) to retrieve libraries in each ecosystem. Similar to He et al. [4], we limit our analysis to only libraries with more than 10 dependent GitHub repositories to exclude the cases where developers upload internal artifacts to Maven, npm, or PyPI but do not expect them to be reused by other projects. After filtering, we get 14,629, 62,051, and 10,147 libraries in Maven, npm, and PyPI, respectively. In this study, we only consider migrations between these libraries.

2) Projects: We utilize the GHTorrent dataset [54] (version 2021-03-06) to retrieve projects in each ecosystem. We select non-fork Python, Java, and JavaScript repositories with more than 10 stars from GHTorrent and get 121,381 Python projects, 66,635 Java projects, and 160,211 JavaScript projects after this step. Similar to previous studies [1], [4], we do not filter by project quality or maturity in order to unveil general facts and longitudinal trends about SALMs. For Java and JavaScript

Algorithm 1: Identifying SALMs

```
Input: Project set $\mathcal{P}$, library set $\mathcal{L}$, and thresholds $t_1, t_2$
Output: A set of SALMs $SALM = \{\langle p, c, l_{src}, l_{tgt}\rangle | p \in \mathcal{P}, c \in \texttt{commits}(p), l_{src} \in \mathcal{L}, l_{tgt} \in \mathcal{L}\}$
$DC \leftarrow \emptyset$  # dependency changes
for $p \in \mathcal{P}$ do
    for $c \in \texttt{commits}(p)$ do
        $dep_{old} \leftarrow \emptyset,\ dep_{new} \leftarrow \emptyset$
        for $\langle file, blob_{old}, blob_{new}\rangle \in \texttt{iter\_diffs}(p, c)$ do
            if not_dependency_file($file$) then
                continue
            $dep_{old} \leftarrow dep_{old} \cup \texttt{extract\_deps}(blob_{old})$
            $dep_{new} \leftarrow dep_{new} \cup \texttt{extract\_deps}(blob_{new})$
        $L^- \leftarrow (dep_{old} - dep_{new}) \cap \mathcal{L}$
        $L^+ \leftarrow (dep_{new} - dep_{old}) \cap \mathcal{L}$
        if $L^- \neq \emptyset \wedge L^+ \neq \emptyset$ then
            $DC \leftarrow DC \cup \{\langle p, c, L^-, L^+\rangle\}$
$SALM \leftarrow \emptyset$  # self-admitted library migrations
for $\langle p, c, L^-, L^+\rangle \in DC$ do
    for $\langle l_{src}, l_{tgt}\rangle \in L^- \times L^+$ do
        if match_commit_message($c, l_{src}, l_{tgt}, t_1$) $\wedge$
          get_rule_frequency($l_{src}, l_{tgt}$) $\geq t_2$ then
            $SALM \leftarrow SALM \cup \{\langle p, c, l_{src}, l_{tgt}\rangle\}$
return $SALM$
```

projects, we limit our analysis to projects with a dependency configuration file (i.e., `pom.xml` and `package.json`, due to reasons explained in Section III-B2), leaving us with 25,289 and 31,768 projects, respectively.

B. Identifying Self-Admitted Library Migrations (SALMs)

To identify SALMs, we design and employ an accurate NLP-powered heuristic-based mining algorithm (Algorithm 1).

1) Overview of the Algorithm: The algorithm takes project set $\mathcal{P}$, library set $\mathcal{L}$, and two threshold parameters t_1 and t_2, as input. It returns a set of (possible) SALMs $SALM$, in which each item is a 4-tuple. Each item $\langle p, c, l_{src}, l_{tgt}\rangle \in SALM$ represents that project p has conducted a SALM from source library l_{src} to target library l_{tgt} in commit c.

The algorithm consists of two main stages. In the first stage (line 1-13), it constructs a set of dependency changes DC in which each $\langle p, c, L^-, L^+\rangle \in DC$ represents that project p removes library set L^- and adds library set L^+ in commit c (ensuring that $L^- \neq \emptyset$, $L^+ \neq \emptyset$, and $L^- \cap L^+ = \emptyset$). To construct DC, it iterates over commits and file diffs (lines 4-5),[2] skips non-dependency files (line 6-7, files other than `pom.xml`, `package.json`, and `*.py` are all skipped in our case), and extracts dependencies from the two file versions[3] (lines 8-9, `extract_deps` will be explained in Section III-B2). Then, a dependency change item $\langle p, c, L^-, L^+\rangle$ is added to DC if the commit contains both added and removed dependencies of our interest (lines 10-13). In the second stage, the algorithm infers SALMs by considering both the commit message and the frequency of the migration rule (line 17), in which

[2]Note that the commits in a git repository form a directed acyclic graph and carefully designed heuristics are needed to avoid duplicate/missed dependency changes (caused by merge commits) during the iteration of commits.

[3]Internally, `git` uses the blob object type to store the content of files in a repository, and each version of a repository file corresponds to one blob

`match_commit_message` will be introduced in Section III-B3 and `get_rule_frequency`(l_{src}, l_{tgt}) is defined as the number of dependency changes in which $\langle l_{src}, l_{tgt} \rangle$ belongs to the Cartesian product of their added and removed libraries.

Note that the algorithm only considers single-commit SALMs, similar to previous work [4], [21], as the accurate detection of multi-commit or non-admitted migrations is much more challenging. In other words, we favor the soundness of our approach (i.e., precision) over broad coverage (i.e., recall) to build a high-quality dataset of actual SALMs.

2) Extracting Dependencies: For Java and JavaScript projects, the `extract_deps()` function takes a lightweight approach and extracts dependencies by parsing their `pom.xml` and `packages.json` files. The rationales are that: 1) Java and JavaScript projects often use a dependency manager (i.e., npm and Maven) and a configuration file (i.e., `pom.xml` and `package.json`) declaring their dependent libraries; 2) their declared dependencies are generally expected to be correct and precise because their configurations are "strict" (i.e., a build will typically fail if dependencies are wrong or missing).

The extraction of dependencies from Python projects is more problematic, as there is no widely adopted dependency management practice in Python. Developers can use various dependency managers (e.g., pip, Conda, Poetry) and dependency configuration files (e.g., `requirements.txt`, `setup.py`, `environment.yml`, `pyproject.toml`); many even do not use a dependency manager [55]. Moreover, several recent studies show that dependency configuration errors are common in Python projects [42], [43], [56]. Therefore, we resort to source code analysis to extract dependencies from the Python project.

For the 10,147 libraries from PyPI, we build a mapping between *library names* (e.g., `scikit-learn`) and *import names* (e.g., `sklearn`). We use a script to download a wheel for each library version and extract top-level import names from wheel metadata, and successfully get import names of 8,715 libraries. For the remaining 1,433 libraries without wheels, we heuristically convert their library names to import names (e.g., convert to lowercase and replace `"-"` with `"_"`). Then, for each project, we get its dependencies by extracting all import statements from Python source code and mapping import names to their corresponding libraries. It is possible that one import name may correspond to more than one library (i.e., import name collisions). In some cases, the import names just happen to collide in two unrelated libraries (e.g., the name `peak` from `SymbolType` and `Importing`); in other cases, a group of related libraries shares the same import name (e.g., the name `azure` from `azure-common`, `azure-mgmt-resource`, and `azure-mgmt-network`). The algorithm considers a project as using all possible libraries in the case of an import name collision, and in the latter case, the collided libraries will be merged into a *super library* (Section III-C).

3) Matching Commit Messages: In the line 17 of Algorithm 1, `match_commit_message`$(c, l_{src}, l_{tgt}, t_1)$ determines whether the commit message of c contains a sentence that states an SALM from l_{src} to l_{tgt}. First, for a commit message, it applies sentence segmentation, word segmentation, part-of-speech tagging, and lemmatization, using NLTK [57], to convert each sentence in the message into word sequences. Second, for each word sequence (i.e. a sentence), it identifies possible words referring to l_{src} or l_{tgt} via fuzzy matching, because developers often do not write canonical library names in commit messages, especially in Java (e.g., `log4j` may stand for `org.apache.logging.log4j:log4j-core`).[4] Third, it searches the word sequence for the presence of any subsequences in Table I (based on the patterns that we summarize from the manually validated migration dataset in He et al. [4]). If the distances of all adjacent words in a matched subsequence are no more than t_1, it will consider the commit message as a match (i.e., likely to be stating a migration from l_{src} to l_{tgt}).

TABLE I. The subsequences for matching commit messages. $[a|b]$ means that both a and b will be matched for this item.

[add|get|use|install|require|switch|move|migrate|port|introduce] l_{tgt}
[remove|drop|delete|abandon] l_{src}
[switch|replace|migrate|swap out|upgrade] l_{src} l_{tgt}
[change|move|update] $[l_{tgt}\ l_{src} \| l_{src}\ l_{tgt}]$
l_{tgt} instead of l_{src}
l_{src} -> l_{tgt}
[update|switch|change|convert|port|replace|require|use|fix] $[l_{src} \| l_{tgt}]$

4) Evaluation: For `match_commit_message()`, we evaluate its effectiveness by computing the recall on the 3,163 migration commits analyzed in He et al. [4] and iteratively refine the function until we believe the results are satisfactory (the final version achieves a recall of 80.71%). For other heuristics and parameters in Algorithm 1, we sample from our identified SALMs, analyze the errors, and iteratively refine the algorithm in a similar manner. Finally, we set $t_1 = 5$ and $t_2 = 2$ and it returns 20,099, 5,926, and 7,642 SALMs from Java, JavaScript, and Python projects, respectively. To evaluate their quality, we sample 377, 360, and 372 SALMs from each (confidence level = 95%, margin error = 5%), manually determine their correctness by reading commit messages and code diffs, and obtain 84.70%, 83.80%, and 86.50% precision, respectively. The inaccuracies mainly come from three cases: 1) collisions between library names and common developer terms (e.g, `sys`, `core`), 2) l_{src} and l_{tgt} share similar names but have different functionalities (e.g., two components from the same framework), and 3) the first two subsequences in Table I can introduce some errors (we retain them to balance between precision and recall). We will handle the second case using *super libraries* (Section III-C).

C. Grouping Super Libraries

It is common for some "groups" of libraries to be closely related to each other. For example, a framework can have several components for different purposes (e.g., the `azure` case in Section III-B2), a library may provide different distributions

[4]More specifically, the function divides the full name of l_{src} and l_{tgt} into small parts (e.g. `org.apache.logging.log4j:log4j-core` is divided into `org`, `apache`, `logging`, `log4j`, and `core`) and fuzzy match these parts in commit message based on Levenshtein distances. Different thresholds are set for parts of different lengths to ensure accuracy. It also handles the case in which the names of l_{src} and l_{tgt} significantly overlap (e.g., a migration from `rollup-plugin-buble` to `@rollup/plugin-buble`).

(e.g., `ant:ant` and `ant:ant-nodeps`), and some libraries may rename themselves in major version updates (e.g., `babel-core` renamed to `@babel/core` since 7.0.0). In some sense, they can be logically considered as a single "library" and studying migrations between them can introduce noises. Therefore, in subsequent analyzes, we manually locate and group libraries from these cases into *super libraries* (e.g., libraries in the three former examples are merged into `azure`, `ant`, and `babel`, respectively). Sometimes, it can be difficult to decide how to group components from a large framework (e.g., `springframework`) into *super libraries*. Since the goal of this step is to reduce the noise caused by migrations within *super libraries* and developers do migrate framework components, we focus on the functionalities of each component and group components with different functionalities into different *super libraries* (e.g. `springdata`, `springcore`, `springcloud`). After grouping, we get 217, 36, and 32 *super libraries* in Java, JavaScript, and Python, respectively.

D. Identifying Domains for SALMs

Considering the size of our dataset, we need to apply some automation to get the application domain of each library for answering **RQ2**. For libraries with natural language descriptions in Libraries.io [53], we leverage BERTopic [58], a robust unsupervised topic modeling technique based on large language models and c-TF-IDF, to get clusters of libraries that may belong to the same application domain. BERTopic [58] can automatically infer the optimal number of clusters and estimate the probability of each library belonging to its cluster. We assign names to each cluster based on the topic words generated by the model and consider all libraries with ≥ 0.5 probability as belonging to that cluster. To verify the reliability of this model, we randomly pick five libraries for each cluster with ≥ 0.5 probability, manually check whether they belong to their clusters, and get an accuracy of 0.93. For the remaining libraries and libraries without descriptions, we search relevant information on the Internet, and manually assign them to existing clusters or introduce a new cluster if necessary. For a migration rule $\langle l_{src}, l_{tgt} \rangle$, we consider it as belonging to domain X if both l_{src} and l_{tgt} are from X. l_{src} and l_{tgt} can also fall into different domains (e.g., from HTTP clients to web frameworks). We merge clusters with a few SALMs into the *Utility* domain and finally get 11, 12, and 16 domains in Java, JavaScript, and Python, respectively.

E. Identifying Rationales for SALMs

Developers may migrate libraries due to various reasons, including usability, features, and so on [4]. An understanding of why they choose to migrate and why they migrate to the target library can help developers select a library or make maintainers aware of the shortcomings of their libraries. Given the scale of our dataset, we choose to first identify, through keywords, possible commit messages that state their rationales and then conduct manual labeling on the filtered dataset.

Similar to Section III-B3, we summarize common keywords for SALM rationales (e.g., "because," "since," "so that") based on the migration dataset retrieved by He et al. [4]. We assume that sentences near the SALM sentence are most likely to mention rationales and only search for the keywords in the SALM sentence and two adjacent sentences. This assumption helps us avoid most false positives and the manual effort of inspecting lengthy commit messages.

TABLE II. An overview of statistics in our dataset

Data/Size	Java	JavaScript	Python
Projects ($\mathcal{P}$)	25,289	31,768	120,321
Libraries ($\mathcal{L}$)	10,147	62,051	14,629
Dependency Changes (DC)	55,797	49,713	361,400
SALMs ($SALM$)	20,099	5,926	7,642
SALMs (grouped)	2,938	1,557	5,805
Migration Rules	2,851	1,032	794
Migration Rules (grouped)	390	259	640
Estimated Precision	84.70%	83.80%	86.50%

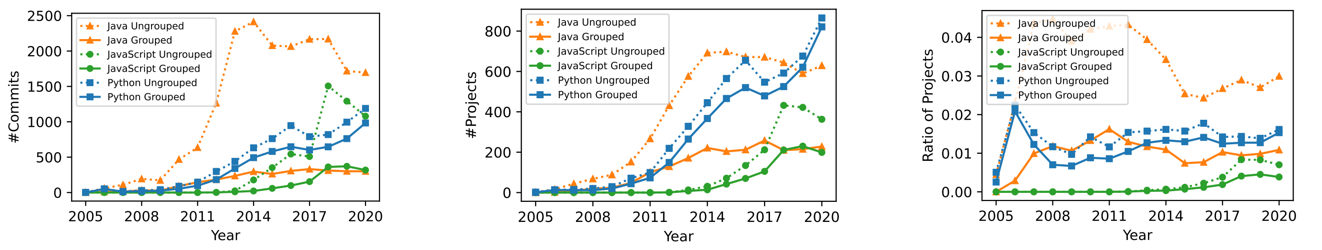
After filtering, we get 582, 283, and 1,126 commit messages from Java, JavaScript, and Python projects, respectively. Then, the first author, with more than three years of software development experience in all three languages, uses the migration rationale framework and the coding book provided by He et al. [4] to code commit messages (including referenced issues and pull requests). If some commit messages cannot fit into the previous framework, he will revise the definitions or add new categories. The second author, with more than five years of software development experience in all three languages, independently uses the new coding book to code all commits and discuss with the first author to resolve the disagreements. The inter-rater reliability between them is 0.83 (Krippendorff's alpha [59], which is a suitable measure because each commit message can be labeled with multiple codes), indicating high agreement [59]. Finally, we classify all commit messages into 13 categories of SALM rationales.

We provide an overview of our dataset in Table II. Here *SALMs (grouped)* and *Migration Rules (grouped)* refer to the number of SALMs and migration rules after the libraries are grouped into super libraries and SALMs within super libraries are removed. In this study, we mainly consider SALMs and migration rules *after grouping*, as they are more likely to be important and non-trivial migrations (as opposed to, e.g., migrations that occur due to library renames).

IV. RESULTS

A. RQ1: How common are SALMs in Java/Maven, JavaScript/npm, and Python/PyPI? How do the longitudinal trends differ in the three ecosystems?

Using Algorithm 1, we get 20,099, 5,926, and 7,642 SALMs in Java, JavaScript, and Python projects, respectively (Table II). Among the studied projects, 3,795 projects (15.01%) in Java, 1,432 projects (4.51%) in JavaScript, and 4,409 projects (3.66%) in Python have conducted at least one SALM. After grouping super libraries, we get 2,938, 1,557, and 5,805 SALMs, respectively. The numbers drop significantly in Java and JavaScript projects, but only marginally in Python projects because more SALMs happen within super libraries in Java and JavaScript projects (mostly due to library renames).



(a) # of SALMs (b) # of active projects with ≥1 SALM(s) (c) Ratio of active projects with ≥1 SALM(s)

Fig. 1: The longitudinal trend of SALMs in Java, JavaScript, and Python projects, from 2005 to 2020

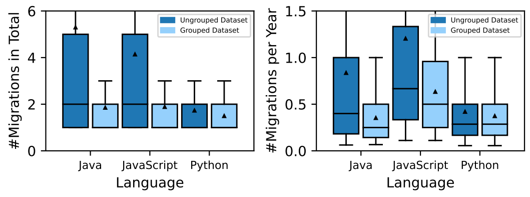


Fig. 2: # of SALMs by a project (in total / per year)

We plot the longitudinal trend of SALMs (with or without grouping) in Figure 1, which starts from 2005 and ends in 2020 because the data are scarce before 2005 and trim in July 2021. Figure 1(a) and 1(b) show that the number of SALMs and the number of active (i.e., having commit activity in this year) projects with ≥1 SALM(s) increase at the beginning but change differently later in the three ecosystems. The ratio of active projects with ≥1 library SALM(s) to all active projects has remained relatively stable with some upward and downward trends (shown in Figure 1(c)).

The number of SALMs and the number of active projects with ≥1 SALM(s) evolve differently in the three ecosystems. For Java, the numbers remain stable and the proportion decreases after 2014; for JavaScript, SALMs emerge late (as npm is first released in 2011), grow rapidly, but have a slight downward trend after 2018; for Python, the number of SALMs is constantly increasing, the proportion fluctuates from 2005 to 2008 (due to the limited number of data points) and shows a slightly increasing trend ever since. Python's peak in 2016 is due to the deprecation of `python-keystoneclient` and `tempest-lib` in all OpenStack projects. The differences indicate that Java and JavaScript projects may have stabilized dependency management practices while Python projects have not reached such stabilization yet, with unsettled library selection best practices and intense competition between libraries.

We compute the number of SALMs in each project and the average number per year, to plot the overall distributions in Figure 2. For each sub-figure, two boxes are drawn for each ecosystem. The left box represents the SALM dataset before grouping super libraries and the right box represents the dataset after grouping. We observe skewed distributions in terms of SALM frequency: 75% projects conduct less than five SALMs in total and one SALM per year. JavaScript projects migrate libraries (0.67 / 0.50 for ungrouped / grouped dataset per year in median) slightly more often than Java (0.40 / 0.25) and Python projects (0.29 / 0.29).

Summary for RQ1:

SALMs are prevalent in all three packaging ecosystems. Java/Maven and JavaScript/npm see a stabilized number of SALMs and a reduced proportion of active projects with ≥1 SALM(s) since 2014 and 2018, respectively. For Python/PyPI, both the number and the proportion show an increasing trend across the observed timespan.

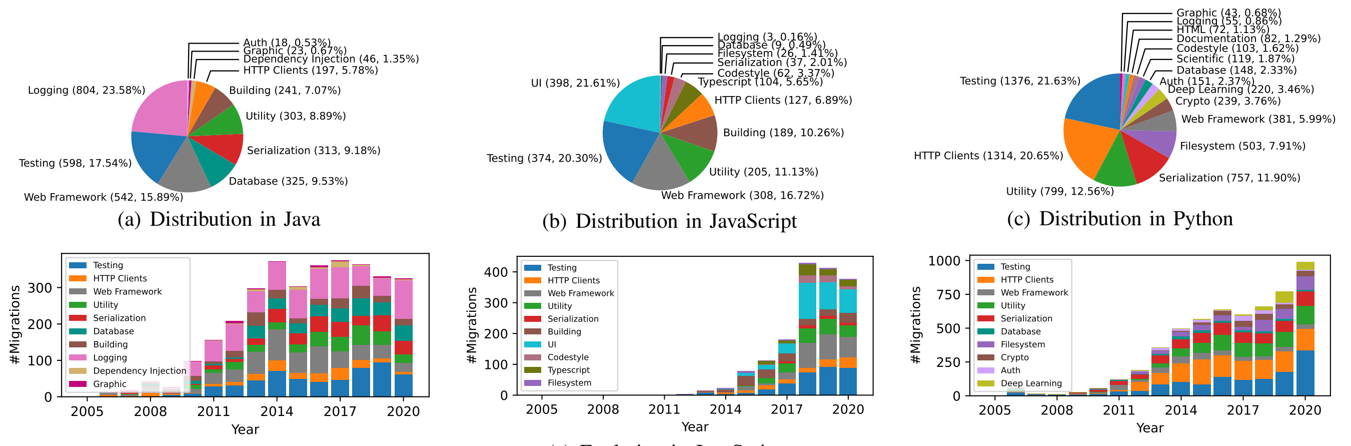
B. RQ2: In what library domains do SALMs happen? How do the domains differ in the three ecosystems?

Following the method in Section III-D, we plot the domain distributions and the evolution of SALMs in each domain for the three ecosystems in Figure 3. To our surprise, although these ecosystems are targeted for different application domains (e.g. frontend for JavaScript), their SALM domains still share some degree of similarity. In all these ecosystems, around half (48.39% in Java, 45.92% in JavaScript, 60.17% in Python) of the SALMs happen among testing frameworks, web frameworks, HTTP clients, and serialization libraries.

The specific distributions differ among the ecosystems and there are also ecosystem-specific domains in which SALMs happen. For example, the percentage of web framework migrations in Java and JavaScript is higher than that in Python. The most prevalent SALM domains are logging libraries in Java (23.58%), UI libraries in JavaScript (21.61%), and testing frameworks in Python (21.63%), two of which have been investigated in prior work [2], [44]. In terms of domain evolution, we find that the number of SALMs among serialization libraries remains stable and the number of SALMs among testing frameworks is constantly increasing in all three ecosystems. Since testing is a sophisticated endeavor and testing frameworks are tightly coupled with the code base, developers tend to migrate to more powerful frameworks with simpler syntax and higher fixture flexibility for ease of maintenance [44] (more details in Section IV-C).

Summary for RQ2:

In all three ecosystems, around half of the SALMs happen among testing frameworks, web frameworks, HTTP clients, and serialization libraries. There are also highly common, yet ecosystem-specific, SALM domains (e.g., UI libraries in JavaScript).



(a) Distribution in Java

(b) Distribution in JavaScript

(c) Distribution in Python

(d) Evolution in Java

(e) Evolution in JavaScript

(f) Evolution in Python

Fig. 3: The distributions of SALM domains and their evolution in Java, JavaScript, and Python projects

C. RQ3: What are the rationales for SALMs? How do the rationales differ in the three ecosystems?

Table III provides detailed definitions and examples for each of the 13 SALM rationale categories, grouped into three themes (source library, target library, and project specific). Rationales whose definitions are extended w.r.t. He et al. [4] are marked with † and the specific differences are highlighted with underlines. In general, we conclude that the previous 13-category framework by He et al. [4] can be generalized to all three ecosystems well, with only some minor modifications.

However, by plotting the distribution of SALM rationales (in terms of the number of projects that have migrated due to the rationale) in Table IV, we observe significant differences among the three ecosystems. For Python projects, properties of the target library are more likely to be the reasons why developers choose to conduct a migration (42.0%, compared with 20.8% and 25.0% in Java and JavaScript projects, respectively). The reason for this may be that the Python/PyPI ecosystem still sees many libraries with similar functionalities competing with each other (e.g., `urllib` and `requests`), and libraries that are more powerful, flexible, or easier to use, can win favor from developers using other libraries. Different from JavaScript, project-specific reasons largely consumes the Java/Maven and Python/PyPI ecosystem (43.0% and 37.8%, compared with 18.5% in JavaScript/npm), indicating that developers in the two ecosystems more often need to integrate different dependencies or fall into various dependency management issues. It is also interesting to note that although both npm and PyPI have a rapidly growing number of libraries, it is more often that issues in source libraries (56.5%, mainly deprecations, 47.2%) trigger migrations in JavaScript projects. In JavaScript/npm, we observe that it is common for new libraries to emerge one after another in the same domain and older libraries can become unmaintained very quickly, causing issues and extra migration workloads for developers. Examples include `node-sass`, `next-sass`, and `dart-sass` frameworks for the implementation of Sass (an extended style sheet language over CSS) and `axios`, `got`, and `request` for making and handling HTTP requests.

We also plot the evolution of SALMs mentioned with rationales in Figure 4. The distribution of SALM rationales is shown more intuitively in these three ecosystems (e.g., the dominating project-specific rationales in Java). Apart from this, we observe a rapidly surging trend of SALMs due to the source library in JavaScript/npm, indicating that the rapid deprecations of libraries are causing more and more issues for JavaScript developers.

Summary for RQ3:

The rationale framework of He et al. [4] is generalizable to all three ecosystems, but the distribution of rationales vastly differ: Java developers tend to migrate for project integration, JavaScript developers tend to migrate due to issues in the source libraries, and Python developers tend to migrate for more competent target libraries.

D. RQ4: Are SALMs unidirectional? How does the directionality differ in the three ecosystems?

Directionality refers to whether a library is always adopted or abandoned by developers among all SALMs. If a migration rule has directionality, or a migration rule is unidirectional, it means that the source library is almost always abandoned and the target library is almost always adopted. Following the study of He et al. [4], we use $flow(l)$ to describe the degree to which a library is adopted or abandoned.

$$flow(l) = \left| \frac{out\_deg(l) - in\_deg(l)}{out\_deg(l) + in\_deg(l)} \right| \tag{1}$$

If $flow(l) = 1$, it means that l is totally abandoned or adopted in all migrations; if $flow(l) = 0$, it means that l has been abandoned and adopted the same number of times.

We plot the distribution of the total SALM flow of these three ecosystems in Figure 5(a). We find that the distribution of $flow(l)$ shows a clear peak no matter in which ecosystem, which means that most libraries are either always adopted or

TABLE III. Definitions and examples of SALM rationales. The rationales marked with † have extended definitions compared to He et al. [4] and the extended definitions are highlighted with underlines.

	Rationale	Definition	Examples
Source Library	Deprecation	The source library is inactive, not maintained, will be deprecated in the future, or not recommended to use officially.	1) *Replace usage of the deprecated `request-promise` library with `axios`* [60] (JavaScript). 2) *Switch deprecated `react-router-redux` to `connected-react-router`* [61] (JavaScript).
	Bug or issue†	1) Source library has bugs or emits warnings. 2) The source library can not be downloaded or installed successfully because of its own issues.	1) *Switched `twitter` back to `tweepy` because the new api was bugging out* [62] (Python). 2) *Changed module to `Cryptodome` to use AES crypto for cpasswords. The `Crypto` package wasn't fetched properly however `Cryptodome` is* [63] (Python).
	Security†	1) The source library has known security vulnerabilities. 2) The source library has security-related issues such as thread insafety and memory leaks.	1) *Switching `rollup` $\rightarrow$ `webpack` for OG image component. Fix `bl` security vulnerability* [64] (JavaScript). 2) *Switch fetch_url from `urllib` to `httplib` to avoid garbage and memleaks* [65] (Python).
Target Library	Functionality†	1) The target library has new and powerful functionalities for project to implement their desired features. 2) The target library is the superset of source library. 3) The target library has more functionalities and make the project more adaptable to future changes.	1) *This includes moving from `underscore` to `lodash` as it has better module support and array functionality* [66] (JavaScript). 2) *Replaced `commentjson` with `PyYAML` (since json is a subset of the yaml format)* [67] (Python). 3) *Replace `pytest-mock` with generic `unittest mock` because `pytest-mock` only supports function-scope* [68] (Python).
	Usability†	1) The target library is easy to install, use, or maintain; the target library has better documentation. 2) Using the target library can bring ease of implementation and cleaner code. 3) The target library provides user-friendly configurations, APIs, and outputs.	1) *Switched to the better documented `assertj`* [69] (Java). 2) *Moved this to use `Gson` because it makes code cleaner* [70] (Java). 3) *Move to `gulp-tasks` flow for (hopefully) easier understanding of tasks* [71] (JavaScript).
	Performance†	1) Using the target library can improve runtime performance. 2) Using the target library results in less resource usage (e.g., memory).	1) *Moved get requests to `aiohttp` for quicker response times* [72] (Python). 2) *Replace `HSQLDB` with `H2` for greatly reduced memory usage* [73] (Java).
	Activity†	Although the source library is under maintenance, the project still chooses to use a more recent, well-maintained dependency.	*`Pydot` seems to be newer than `pydotplus`, and has more maintainers. So switch this code to use `pydot`* [74] (Python).
	Popularity	The target library is more widely used or complies with industrial standards/ecosystem best practices.	*`GSON` is used by many upstream ODL projects and is the desired single JSON library for the future* [75] (Java).
	Size/Complexity	The target library is simpler or results in a smaller binary size.	*This swaps `chalk` out for `kleur` which is smaller and faster.* [76] (JavaScript).
Project Specific	Integration	1) The project needs to integrate with operating systems, platforms, or frameworks. 2) The project needs to support a new version or standard of programming language (Python 3, ES 6, Java 8, etc.). 3) The project has to resolve conflicts between dependencies.	1) *Uses `apache httpcomponents` instead of `commons httpclient` (so that it can run on an android device without other dependencies for example)* [77] (Java). 2) *Use `uglify-es` for ES6 support* [78] (JavaScript). 3) *`Mockito` and `jMock` pulls in the different versions of `Hamcrest`, and it conflicts with the version pulled by `jUnit`* [79] (Java).
	Simplification†	1) The project migrate to reduce the number of dependencies. 2) The project unifies dependencies to ensure consistency within the project and avoid duplications. 3) The project no longer needs to support legacy platforms (e.g., Python 2) and replace relevant libraries.	1) *Use built-in `json` for `object-to-JSON` conversion* [80] (Python). 2) *Replaced `SLF4J` and `Logback` with `log4j` bridge, to unify log configuration* [81] (Java). 3) *It's 2020, and we've long deprecated py26, so we can rely on `argparse` to be available* [82] (Python).
	Organization	The project conducts a migration to follow the conventions or regulations of its belonging community or organization.	*In order to migrate the code base to the "OpenStack way"* [83] (Python).
	License	The project conducts migration to resolve incompatibilities between open-source licenses.	*Replaced the `org.json` JSON lib with `JSON.simple` since the license of `org.json` isn't GPL compatible* [84] (Java).

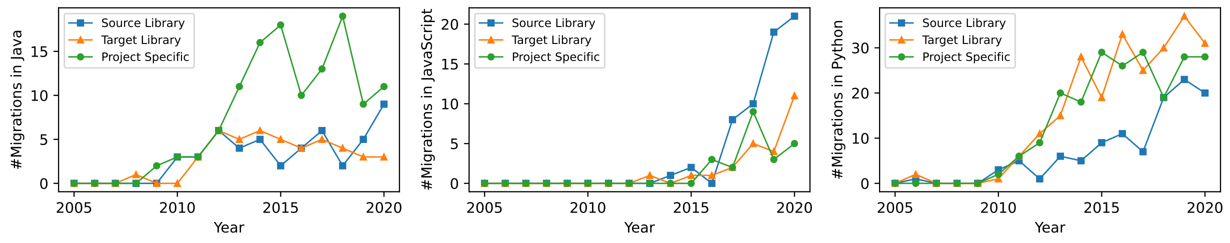


Fig. 4: # of projects that migrate due to different rationales in each year in Java, JavaScript, and Python

abandoned in SALMs. However, it is worth noting that there are more libraries whose adoption rate and abandonment rate are similar ($flow(l)$ are closer to 0) in Python than in Java and JavaScript, such as `unittest`, `Flask`, `urllib` and so on, which means developers in different projects can either adopt or abandon these libraries.

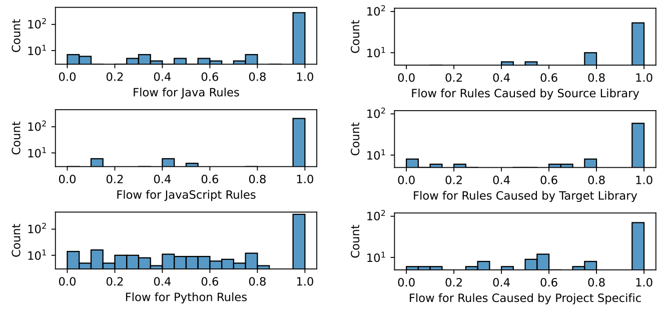
To explore further, we seek to find the relationship between the directionality and the SALM rationales summarized in Section IV-C. Therefore, we calculate the SALM flows for libraries migrated due to source library, target library, and project-specific, respectively, and plot their distributions in Figure 5(b). The results show that the distribution of SALM flow seems more uniform (more libraries' $flow(l)$ are closer to 0) for libraries migrated due to features of the target library or project-specific reasons than that for libraries migrated due to problems in the source library. It means that SALMs because of source libraries are more unidirectional than those because of target libraries and project-specific reasons. Most developers will drop a deprecated or buggy library, but different developers have different criteria for functionality, usability, and performance. SALMs caused by project-specific reasons are also less unidirectional as projects may have different contexts and use cases where the most suitable library may differ.

Summary for RQ4:

SALMs are highly unidirectional in all three ecosystems with a less degree in Python/PyPI. SALMs due to issues in the source library tend to be more unidirectional than SALMs due to target library or project-specific reasons.

TABLE IV. Distribution of SALM rationales

Rationale	Java		JavaScript		Python	
Source Library	46	22.2%	61	56.5%	107	20.2%
Deprecation	35	16.9%	51	47.2%	56	10.6%
Bug or issue	6	2.9%	7	6.5%	37	7.0%
Security†	5	2.4%	3	2.8%	14	2.6%
Target Library	43	20.8%	27	25.0%	222	42.0%
Feature	13	6.3%	10	9.3%	88	16.6%
Usability†	18	8.7%	4	3.7%	58	11.0%
Performance†	6	2.9%	6	5.6%	61	11.5%
Activity†	3	1.4%	2	1.9%	8	1.5%
Popularity	2	1.0%	2	1.9%	6	1.1%
Size/Complexity	1	0.5%	3	2.8%	1	0.2%
Project Specific	89	43.0%	20	18.5%	200	37.8%
Integration†	54	26.1%	17	15.7%	146	27.6%
Simplification†	33	15.9%	3	2.8%	48	9.1%
Organization Influence	0	0.0%	0	0.0%	3	0.6%
License	2	1.0%	0	0.0%	3	0.6%
Total	178	100.0%	108	100.0%	529	100.0%



(a) Distribution of total $flow(l)$ (b) Distribution of $flow(l)$ migrated due to different reasons

Fig. 5: $flow(l)$ distributions in the three ecosystems

V. Discussion

A. Implications

The results from our study indicate that, despite the high cost, a non-negligible number of projects have conducted library migration in all three packaging ecosystems. Most of the migrations happen between "competitor" libraries in the same domain, and it is common for projects to mistakenly select one library and later migrate to another, indicating flaws in the current library selection practices. Although several tools have been proposed recently to support library selection [29], [85], they often do not consider post-selection failures. It would be interesting to investigate why developers overlook important library properties (like those in Table III) at the beginning and design tools to overcome the limitations of current practice.

There are both universal library domains in which migrations frequently happen (i.e., testing frameworks, web frameworks, HTTP clients, and serialization) and domains unique to each ecosystem (e.g., UI libraries in JavaScript/npm). Similar to previous studies on Java logging library migrations [2] and Python testing framework migrations [44], we suggest future in-depth studies on other popular domains (e.g., serialization, UI, HTTP clients) to reveal more information on these migrations and formulate domain-specific best practices. Our dataset (see Section VII) can serve as a starting point.

1) The Python/PyPI Ecosystem: It is noteworthy that library migrations in Python/PyPI maintain stable growth in the past decade (**RQ1**), which contrasts sharply with the stabilized and slightly decreasing trend of Java/Maven and JavaScript/npm. Moreover, Python projects are generally more concerned with the overall superiority of target libraries (**RQ3**), and the migrations show less unidirectionality compared with Java and JavaScript projects (**RQ4**). The unique characteristics of Python/PyPI indicate that it is in a critical stage of development: libraries with different features and technologies are constantly emerging and competing for the state-of-the-art. However, as indicated by the presence of SALMs and results from recent studies [42], [43], [55], [56], Python developers are facing a multitude of dependency management problems. To help practitioners select and migrate libraries, we suggest researchers investigate library recommendation and API migration approaches in Python/PyPI, similar to previous studies in the Java literature [21], [48], [50].

2) On the Development of Packaging Ecosystems: From the perspective of library migration, we argue that JavaScript/npm, Python/PyPI, and Java/Maven represent three different stages of packaging ecosystem development, which we name here as *the stage of retention*, *the stage of competition*, and *the stage of evolution & expansion*, respectively.

With only a decade of history (as npm was first released in 2011), JavaScript/npm seems to be currently in *the stage of retention*. In this stage, a large number of libraries are released every day and their main goal is to ensure their retention and adoption by developers. Therefore, JavaScript/npm libraries are generally of smaller scale (many of them are even trivial packages [24]) and evolve rapidly, but these libraries are also more likely to have quality, security, or sustainability issues. This may explain why in JavaScript/npm, most of the SALMs happen due to issues in the source library (**RQ3**). Although the deprecation mechanism in npm [86] helps developers avoid and migrate deprecated libraries, it may be more useful to have mechanisms to signal libraries in decline early [87].

For Python/PyPI, it is likely at an intermediate stage of development, *the stage of competition*. In this stage, developers are more concerned about the key features (i.e., functionality, usability, performance) of libraries. Technology innovation is common and there are often several libraries in one popular domain with similar functionalities. However, these libraries have different key features and developers are still unaware of any certain best practices for library selection. This is the stage where utilities for comparing similar libraries can be especially helpful [28], [29], [85], e.g., a platform that groups and labels libraries by their functionalities, outlines their key features, and demonstrate their key metrics for facilitating adoption and migration (e.g., recent adoption rates, migration flows).

For the Java/Maven ecosystem, the most mature ecosystem of the three, it has undergone the former two stages and entered *the stage of evolution & expansion*. In this stage, libraries with the best key features have won and stood firm. The main task for maintainers is to maintain and evolve their libraries (i.e., upgrades). During library migrations, developers' concerns gradually transfer to simplification and integration with their projects (**RQ3**). To reduce this effort, we suggest developers use related automated tools (e.g., Maven helper [88]) to check and resolve conflicts between dependencies and licenses.

Whatever stage a packaging ecosystem has reached, the "life and death" of libraries and the migrations between libraries are inevitable due to the decentralized and voluntary nature of open source development. We believe that the tools and strategies mentioned above would be very important in both the dependency management of individual projects and the sustainability of packaging ecosystems.

B. Threats to Validity

1) Construct Validity: We use SALMs as the proxy for studying all library migrations, which may result in the omission of information to a certain extent. As SALMs can only be identified from projects with good commit messages, our intuition is that they may be of higher quality and more valuable for practice. It is also difficult to obtain a comprehensive *non-admitted* migration dataset and thus estimate the ratio of SALMs compared with all actual library migrations that happened in the wild. Despite this, SALMs alone have already provided a promising landscape and we leave the investigation of non-admitted migrations for future work.

2) Internal Validity: It is non-trivial implementation work to compute dependency changes from git repositories and any implementation errors would more or less introduce noise into our dataset. To mitigate this threat, we carefully design and test a strict data structure that ensures each dependency change is included once and only once. Extracting dependencies from configuration files and source code brings limitations as well: dependencies declared in configuration files or source code files can be unused (i.e., bloated). However, our focus on SALMs can guarantee the high accuracy of our mining approach (Algorithm 1) and mitigate this effect. We also drop duplicate migrations to avoid overestimation because of dependency transfer between different modules or hidden forks. The performance of our mining approach significantly influences the soundness of our results. Therefore, we iteratively refine the algorithm based on our dataset and try out different thresholds to identify the optimal approach that balances between precision and recall. Filter reasons by keywords may cause a loss of information to a certain extent, so we adopt as many related words as possible based on a previous dataset [4]. The identification of SALMs, the group of *super libraries*, the resolution of library domains, and the categorization reasons may all have labeling or manual errors. Thus, the two authors double check the accuracy, gather reliable information, and code independently to alleviate this threat. We mainly considered SALMs after the libraries are grouped into *super libraries*. On the one hand, grouping *super libraries* can mitigate noisy migrations between different components or distributions of the same framework. On the other hand, it can avoid trivial migrations due to library renames. We believe the grouped dataset can bring us more accurate and in-depth insight into library migrations.

3) External Validity: Our findings may not generalize to migrations in other projects and between other libraries. We mitigate this threat by selecting a large-scale dataset of GitHub projects and popular libraries. Our findings may not generalize to projects that do not use PyPI, Maven, or npm, and migrations beyond these packaging ecosystems. However, the three languages and their packaging ecosystems are the most popular ones on GitHub, are adopted in many different application domains, and demonstrate different characteristics in our results. Therefore, we believe the findings based on the three ecosystems can provide insights applicable to a diverse range of developers and stakeholders.

VI. Conclusion

In this paper, we report a comparative study that mines and compares a massive number of SALMs from GitHub repositories depending on libraries from the three software packaging ecosystems: Java/Maven, JavaScript/npm, and Python/PyPI. To summarize, this paper makes the following main contributions:

- A mining algorithm and a semi-automatic method that accurately finds SALMs and their corresponding rationales from git repositories.
- An empirical verification on the generalizability of previous library migration research in the Java/Maven ecosystem [1]–[3], especially [4].
- A set of commonalities and discrepancies regarding how and why SALMs happen in the three ecosystems, which points to several future research directions and reveals insights about the development of software packaging ecosystems in general.

Several directions of future work arise from our study. The first one is the in-depth case studies of library migrations in the common application domains revealed by our study. The second one is the development of novel library recommendation and API migration approaches in the Python/PyPI and JavaScript/npm ecosystems. Finally, it is also worth investigating the general theories of how libraries compete with each other and how a packaging ecosystem is formed.

VII. Data Availability

The replication package of our study is available at:

https://doi.org/10.5281/zenodo.7524376
https://github.com/guhaiqiao/SALMC

It contains a preprocessed and curated dataset of library migrations in the Java/Maven, JavaScript/npm, and Python/PyPI packaging ecosystems. It also contains the Python scripts and Jupyter Notebooks which can replicate the results of all four research questions in our paper. We hope the replication package can be used to facilitate further research on library migrations and other related topics.

Acknowledgment

This work is supported by the National Natural Science Foundation of China Grant 61825201 and 62142201. We sincerely thank Runzhi He for his suggestions and contributions to dataset curation, and the anonymous reviewers for their insightful and constructive comments.

References

[1] C. Teyton, J. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in Java," *J. Softw. Evol. Process.*, vol. 26, no. 11, pp. 1030–1052, 2014.

[2] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the Apache Software Foundation projects," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016, pp. 154–164.

[3] H. Alrubaye, D. Alshoaibi, E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "How does library migration impact software quality and comprehension? An empirical study," in *Reuse in Emerging Software Engineering Practices - 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2-4, 2020, Proceedings*, ser. Lecture Notes in Computer Science, vol. 12541. Springer, 2020, pp. 245–260.

[4] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on Java library migrations: Prevalence, trends, and rationales," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 478–490.

[5] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: A review of industrial studies," *Empir. Softw. Eng.*, vol. 12, no. 5, pp. 471–516, 2007.

[6] (2022, Augest) Module counts. [Online]. Available: http://www.modulecounts.com/

[7] (2022, Augest) Maven. [Online]. Available: https://mvnrepository.com/

[8] (2022, Augest) npm. [Online]. Available: https://www.npmjs.com/

[9] (2022, Augest) PyPI. [Online]. Available: https://pypi.org/

[10] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 381–416, 2019.

[11] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 2019, pp. 995–1010.

[12] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in Java projects," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 35–45.

[13] I. Pashchenko, D. L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. ACM, 2020, pp. 1513–1531.

[14] E. L. Vargas, M. F. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 245–256.

[15] R. G. Kula, D. M. Germán, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? - An empirical study on the impact of security advisories on library migration," *Empir. Softw. Eng.*, vol. 23, no. 1, pp. 384–417, 2018.

[16] F. R. Côgo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2457–2470, 2021.

[17] M. Valiev, B. Vasilescu, and J. D. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 644–655.

[18] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, 2018, pp. 181–191.

[19] GitHub, Inc. (2021) The state of the Octoverse: Top languages over the years. [Online]. Available: https://octoverse.github.com/#top-languages-over-the-years

[20] C. Teyton, J. Falleri, and X. Blanc, "Mining library migration graphs," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 2012, pp. 289–298.

[21] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 2021, pp. 72–83.

[22] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in Q&A discussions - Incorporating relational and categorical knowledge into word embedding," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 338–348.

[23] D. Kavaler, A. Trockman, B. Vasilescu, and V. Filkov, "Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 476–487.

[24] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? An empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 2017, pp. 385–395.

[25] A. Pano, D. Graziotin, and P. Abrahamsson, "Factors and actors leading to the adoption of a JavaScript framework," *Empir. Softw. Eng.*, vol. 23, no. 6, pp. 3503–3534, 2018.

[26] L. Yin and V. Filkov, "Team discussions and dynamics during DevOps tool adoptions in OSS projects," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 697–708.

[27] Y. Ma, A. Mockus, R. Zaretzki, R. V. Bradley, and B. C. Bichescu, "A methodology for analyzing uptake of software technologies among developers," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 485–501, 2022.

[28] Y. Huang, C. Chen, Z. Xing, T. Lin, and Y. Liu, "Tell them apart: Distilling technology differences from crowd-scale comparison discussions," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 214–224.

[29] R. E. Hajj and S. Nadi, "LibComp: An IntelliJ plugin for comparing Java libraries," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 1591–1595.

[30] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming over Time*. O'Reilly Media, 2020.

[31] Sonatype, Inc. (2021, January) State of the software supply chain. [Online]. Available: https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021

[32] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *J. Syst. Softw.*, vol. 129, pp. 140–158, 2017.

[33] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: An evolutionary study," *Empir. Softw. Eng.*, vol. 20, no. 5, pp. 1275–1317, 2015.

[34] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 84–94.

[35] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of Dependabot security pull requests," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, 2021, pp. 254–265.

[36] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on GitHub Dependabot," *CoRR*, vol. abs/2206.07230, 2022.

[37] J. Henkel and A. Diwan, "CatchUp!: Capturing and replaying refactorings to support API evolution," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM, 2005, pp. 274–283.

[38] K. Huang, B. Chen, L. Pan, S. Wu, and X. Peng, "REPFINDER: Finding replacements for missing APIs in library update," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 266–278.

[39] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 319–330.

[40] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for Python library ecosystem," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 125–135.

[41] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, "Interactive, effort-aware library version harmonization," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, 2020, pp. 518–529.

[42] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of Jupyter notebooks," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1622–1633.

[43] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*. ACM, 2021, pp. 439–451.

[44] L. Barbosa and A. C. Hora, "How and why developers migrate Python tests," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 538–548.

[45] M. Islam, A. K. Jha, and S. Nadi, "PyMigBench and PyMigTax: A benchmark and taxonomy for Python library migration," *CoRR*, vol. abs/2207.01124, 2022.

[46] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm, "Study of an API migration for two XML apis," in *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 5969. Springer, 2009, pp. 42–61.

[47] T. T. Bartolomei, K. Czarnecki, and R. Lämmel, "Swing to SWT and back: Patterns for API migration by wrapping," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010, pp. 1–10.

[48] C. Teyton, J. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 192–201.

[49] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "On the use of information retrieval to automate the detection of third-party Java library migration at the method level," in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 347–357.

[50] C. Chen, Z. Xing, Y. Liu, and K. O. L. Xiong, "Mining likely analogical APIs across third-party libraries via large-scale unsupervised API semantics embedding," *IEEE Trans. Software Eng.*, vol. 47, no. 3, pp. 432–447, 2021.

[51] H. He, Y. Xu, X. Cheng, G. Liang, and M. Zhou, "MigrationAdvisor: Recommending library migrations from large-scale open-source data," in *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 9–12.

[52] M. Lamothe, Y. Guéhéneuc, and W. Shang, "A systematic review of API evolution literature," *ACM Comput. Surv.*, vol. 54, no. 8, pp. 171:1–171:36, 2022.

[53] J. Katz, "Libraries.io Open Source Repository and Dependency Metadata," Jan. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3626071

[54] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*. IEEE Computer Society, 2012, pp. 12–21.

[55] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of Jupyter notebooks," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, 2019, pp. 507–517.

[56] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, "Towards better dependency management: A first look at dependency smells in python projects," *IEEE Transactions on Software Engineering*, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2022.3191353

[57] S. Bird, "NLTK: The natural language toolkit," in *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, Sydney, Australia, 17-21 July 2006*. The Association for Computer Linguistics, 2006.

[58] M. Grootendorst, "BERTopic: Neural topic modeling with a class-based TF-IDF procedure," *CoRR*, vol. abs/2203.05794, 2022.

[59] K. Krippendorff, "Computing krippendorff's alpha-reliability," 2011.

[60] https://github.com/fusionjs/fusionjs/commit/bfd76bb.

[61] https://github.com/quiltdata/t4/commit/97798e7.

[62] https://github.com/CloudBotIRC/CloudBot/commit/f824322.

[63] https://github.com/CasperGN/ActiveDirectoryEnumeration/commit/b5a2f99.

[64] https://github.com/livecoders/website/issues/100.

[65] https://github.com/mknx/smarthome/commit/5a272ab.

[66] https://github.com/Kaetram/Kaetram-Open/commit/26598ef.

[67] https://github.com/DualSpark/cloudformation-environmentbase/commit/d9c06cf.

[68] https://github.com/WhatsApp/WADebug/commit/ded87cb.

[69] https://github.com/fabric8io/fabric8/commit/a2330a1.

[70] https://github.com/Discord4J/Discord4J/commit/9dec9b9.

[71] https://github.com/manshar/manshar/commit/9eb0a3e.

[72] https://github.com/Roxxers/roxbot/commit/776585c.

[73] https://github.com/theotherp/nzbhydra2/commit/76b33c7.

[74] https://github.com/mathics/Mathics/commit/915daeb.

[75] https://github.com/opendaylight/aaa/commit/679bb6d.

[76] https://github.com/FredKSchott/create-snowpack-app/pull/174.

[77] https://github.com/migtavares/owmClient/commit/1e42454.

[78] https://github.com/NodeBB/NodeBB/commit/a00f1f9.

[79] https://github.com/netty/netty/commit/96d5968.

[80] https://github.com/themill/wiz/commit/ba2bf7f.

[81] https://github.com/hazelcast/hazelcast-simulator/commit/9ec60be.

[82] https://github.com/hibtc/cpymad/commit/cad4f01.

[83] https://github.com/openstack-archive/almanach/commit/e056127.

[84] https://github.com/ashleyj/aura/commit/0e0e589.

[85] L. Yan, M. Kim, B. Hartmann, T. Zhang, and E. L. Glassman, "Concept-annotated examples for library comparison," in *The 35th Annual ACM Symposium on User Interface Software and Technology, UIST 2022, Bend, OR, USA, 29 October 2022 - 2 November 2022*, M. Agrawala, J. O. Wobbrock, E. Adar, and V. Setlur, Eds. ACM, 2022, pp. 65:1–65:16. [Online]. Available: https://doi.org/10.1145/3526113.3545647

[86] F. R. Côgo, G. A. Oliva, and A. E. Hassan, "Deprecation of packages and releases in software ecosystems: A case study on NPM," *IEEE Trans. Software Eng.*, vol. 48, no. 7, pp. 2208–2223, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3055123

[87] S. Mujahid, D. E. Costa, R. Abdalkareem, E. Shihab, M. A. Saied, and B. Adams, "Toward using package centrality trend to identify packages in decline," *IEEE Trans. Engineering Management*, vol. 69, no. 6, pp. 3618–3632, 2022. [Online]. Available: https://doi.org/10.1109/TEM.2021.3122012

[88] Maven Helper. [Online]. Available: https://plugins.jetbrains.com/plugin/7179-maven-helper