

Graduation Thesis

Title

Comparing Bug-Fix Commits Between PyPI and Maven: A Study of Cross-Ecosystem Libraries

Supervisor

Professor Kula Raula Gaikovina

Author

Kaya Kitada

February 9th, 2026

Department of Information and Computer Sciences
School of Engineering Science The University of Osaka

Comparing Bug-Fix Commits Between PyPI and Maven: A Study of Cross-Ecosystem Libraries

Kaya Kitada

Abstract

In modern software development, cross-ecosystem libraries allow developers to use the functions they need while choosing their preferred programming language. Users expect these libraries to offer the same features regardless of the language interface they use. However, in practice, some languages such as Java and Python are managed as independent ecosystems. This creates a risk where a bug fixed in one language may not be propagated to the other implementation. The purpose of this study is to empirically clarify how bug fixes are performed and propagated between Java and Python libraries.

This study focuses on libraries maintained in both Java and Python languages distributed via Maven and PyPI. For each commit, we analyzed the relationship between commit messages and modified files. We investigated whether bug fixes were contained within a single language or if they related to changes across language boundaries.

Our analysis shows that simultaneous bug fixes in both Java and Python within a single commit depend on the project's implementation structure. In projects with Runtime Interdependency, simultaneous fixes occur relatively often to maintain consistency. In contrast, in projects with Independent Implementation, such fixes are rare, and modifications are usually handled separately for each language. Interestingly, we found cases where maintainers mentioned the other language in commit messages even when maintainers are aware of cross-ecosystem impacts even during separate fixes. Additionally, the types of fixes show language-specific trends: Java fixes are dominated by build settings and environments, while Python fixes focus on external library updates.

This study analyzes the extent to which bug fixes are reflected and propagated across languages in cross-ecosystem libraries provided in both Java and Python. Our commit-level analysis shows that simultaneous fixes in a single commit are extremely rare. However, the patterns vary by bug type: while build infrastructure and evolution fixes are often limited to one language, runtime errors frequently involve mentions of both languages. This suggests that maintainers are highly aware of consistency between languages regarding runtime safety. These findings help users select

a more suitable library by checking if fixes are properly reflected or if one language is being neglected. Specifically, users can judge the library's quality based on the types of bugs that occur. Future work is needed to further clarify the propagation mechanism through source code difference (diff) analysis and timing analysis of fixes.

Keywords

Cross-ecosystem library

Bug-fix propagation

Contents

1	Introduction	5
2	Related Work	7
2.1	Existing Studies on Cross-Ecosystem Libraries	7
2.2	Potential Maintenance Burden in Multi-Language Projects	7
2.3	Research Contributions	8
3	Dataset Extraction	9
3.1	Data Source	9
3.2	Overview of Data Collection Process	10
3.2.1	Step 1: Extracting Common Repository URLs	10
3.2.2	Step 2: Verification of Project Structure and Selection	11
3.2.3	Step 3: Commit History Extraction and Data Structuring	12
4	Approach	13
4.1	RQ1: To what extent do bug-fix commits from dual-ecosystem libraries propagate to both code bases (Python and Java)?	13
4.1.1	Extracting Bug-Fix Commit	14
4.1.2	Map Fixes to Programming Language	14
4.1.3	Classify Maintenance Balance	15
4.2	RQ2: Are there different to the kinds of bug fixes that propagate to the dual ecosystems?	15
4.2.1	Mining Commit Messages	15
4.2.2	Taxonomy of Bug-Fix Commit Messages	16
4.2.3	Analysis Perspectives	17
5	Results	18
5.1	RQ1: Distribution of Bug-Fix Activities by Language	18
5.1.1	Classification of Projects Based on Metrics	18
5.1.2	Simultaneous Maintenance Rates by Repository	19
5.1.3	Overlap of Detection Metrics	20
5.2	RQ2 Relationship Between Technical Factors and Languages	20
5.2.1	Language Mention Distribution per Technical Factor (Perspective A)	20
5.2.2	Technical Factor Distribution per Language Category (Perspective B)	21

6	Discussion	22
6.1	RQ1: Discussion on Maintenance Balance in Cross-Ecosystem Projects	22
6.1.1	Selection of Repositories for Comparative Analysis	22
6.1.2	Classification of Bug Fix Trends Based on Implementation Structure . . .	22
6.1.3	Intent Revealed by Mismatches Between File Fixes and Language Mentions	23
6.1.4	The Meaning of Commits Without Language Mentions	23
6.2	RQ2 Analysis Results: Qualitative Differences in Maintenance Between Ecosystems	24
6.2.1	Type A: Build and Infrastructure Fixes Concentrated in Java	24
6.2.2	Type D: Evolution and Specification Fixes Concentrated in Python	24
6.2.3	Type C: Shared Responsibility in Runtime Error Fixes	25
6.2.4	Limitations	25
7	Conclusion and Future work	26
7.1	Research Summary	26
7.1.1	Separation of Fix Units in Maintenance Activities (RQ1)	26
7.1.2	Differences in Maintenance Content Between Ecosystems (RQ2)	26
7.2	Future Work	27
7.2.1	Future Work 1: Improving Analysis Methods: Source Code Diff Analysis .	27
7.2.2	Future Work 2: Analyzing Bug Impacts in Wrapper Structures	27
7.2.3	Future Work 3: Quantitative Analysis of Fix Timing and Delays	28
7.2.4	Future Work 4: Expanding the Dataset and Target Languages	28
	Acknowledgement	29
	References	30

1 Introduction

In modern software development, developers implement functions efficiently by leveraging existing libraries. Today, the use of libraries goes beyond simple functional integration; projects are supported by complex chains of multi-layered dependencies, where a project depends on a specific library, which in turn requires multiple other libraries. In most cases, developers introduce these libraries through language-specific package managers. For instance, library distribution and dependency management are conducted within independent ecosystems for each language, such as PyPI for Python and Maven Central for Java. During the development process, projects may undergo a transition from one programming language to another, or a gradual migration of the codebase from an existing language to a newer one. In such situations, developers search for implementations in the target language’s ecosystem that offer the same functions as the libraries they previously used. However, even a library that is widely popular in one language does not necessarily have an equivalent implementation in another. Furthermore, it is uncommon to introduce runtimes or package managers from different languages in parallel just to use a specific library, as this increases the complexity of operational management. Against this background, a common practice has emerged where library maintainers themselves implement the same functions for multiple languages and distribute them across their respective ecosystems. In this study, based on related work [1], we define libraries provided to multiple ecosystems in this manner as cross-ecosystem libraries.

Quantitative studies have already begun to identify the prevalence of cross-ecosystem libraries. Previous research [6] has shown that thousands of repositories manage implementations for multiple languages—such as Java and Python—within a single GitHub repository, sharing the same commit history and issue tracker. While development activities are centralized, existing analysis [9] of these shared repositories has relied mostly on externally observable metrics, such as the number of Issues or Pull Requests. These metrics do not distinguish between specific fix contents or target languages. Therefore, they are insufficient for evaluating how bug fixes propagate between languages or how this affects product quality. There is still much to understand about how managing multiple languages in one repository actually impacts users.

A potential risk in cross-ecosystem libraries is that a bug fixed in one language implementation might remain unpatched in another. Macro-level metrics, such as the total number of commits or issues, cannot identify which language received which fix. This makes it extremely difficult to evaluate the quality and reliability of the entire library. The importance of focusing on bug fixes is rooted in practical library selection. Research by Larios et al. [8] indicates that maintainers con-

sider fix history and bug response as direct factors in judging reliability and maintenance costs. If bug fixes are not applied to all language implementations without delay, users may unintentionally use outdated versions containing vulnerabilities. Furthermore, reports show that many users do not notice update notifications, and even when they do, only a few actually apply the fixes [5]. Other studies point out that many packages are neglected even after their support ends [10]. These facts suggest that maintainers do not always apply fixes to all language implementations immediately. Thus, bug fix propagation is a critical but invisible challenge that cannot be captured by simple activity metrics.

The objective of this study is to empirically clarify the extent to which bug fixes are reflected and propagated across languages in cross-ecosystem libraries. Specifically, we identify cases where a bug fixed in one language is not reflected in the other and analyze these trends. We define the following two Research Questions:

- **RQ1:** To what extent do bug-fix commits from dual-ecosystem libraries propagate to both code bases (Python and Java)?
- **RQ2:** Are there different to the kinds of bug fixes that propagate to the dual ecosystems?

We perform a detailed analysis of commit histories for both languages, treating individual bug-fix commits as the primary unit of analysis. By using techniques like topic mining —which automatically extracts underlying themes and categories from commit messages— to classify the technical nature of these fixes, we reveal qualitative characteristics that traditional statistical data cannot capture. The contribution of this study is the redefinition of cross-ecosystem libraries as subjects for evaluating bug fix consistency. Our findings will assist users in making more informed decisions when selecting libraries, based on an understanding of bug-fix propagation patterns and common bug types.

2 Related Work

2.1 Existing Studies on Cross-Ecosystem Libraries

Software development ecosystems are built on language-specific package managers such as npm, PyPI, and Maven. A prior study [6] mapped metadata from over 1.1 million GitHub projects to identify libraries provided across multiple ecosystems. This study showed that many cross-ecosystem libraries are highly popular and widely used across different programming communities. Most of these libraries manage implementations for multiple languages in a single GitHub repository. This means they share development history (commits) and bug management (Issue trackers). These findings show that cross-ecosystem libraries are important for practical software development. Furthermore, managing multiple implementations in one repository is a common practice in practice. Other studies also identify cross-ecosystem libraries based on shared GitHub repositories. Constantinou et al. [1] analyzed libraries across 12 major ecosystems using metadata from libraries.io. They found that while cross-ecosystem libraries are a minority, they often have high popularity and high dependency levels. Therefore, they tend to have a broader impact than libraries limited to a single ecosystem. While previous studies clarified the prevalence and structure of these libraries, their analysis was limited to the package or release level. Recent study [2] has shown that even when the same semantic version number is used across ecosystems, the actual bug fixes included in each release may differ due to asynchronous release timing. As a result, implementation-level behavior, such as how individual bug fixes are propagated and shared across different language implementations, remains insufficiently understood.

2.2 Potential Maintenance Burden in Multi-Language Projects

The maintenance burden of multi-language development has primarily been analyzed through repository-level activity logs. A previous study [9] compared the frequency of Issues and Pull Requests before and after a project expanded across ecosystems. The results showed that, except for a short period after the expansion, total activity levels remained similar to non-expanded projects. This suggests that managing multiple implementations in one repository is sustainable and does not create an excessive workload. However, stable activity at the repository level does not guarantee that bug fixes are applied equally to every language. Existing research on activity metrics might not fully reflect the user’s perspective. Even in single-ecosystem studies, users often continue to use libraries with unpatched bugs or stopped updates without realizing it [7, 10, 12]. Furthermore, even with automated fix suggestions, many fixes are never merged due to notifica-

tion overload [11]. Empirical studies in software engineering show that when selecting libraries, developers consider attributes such as active issue resolution and maintenance continuity as key factors for long-term reliability [8]. Common metrics like "stars" or download counts can also lead to wrong conclusions because they only reflect specific aspects of a project [15]. Therefore, evaluating cross-ecosystem libraries based only on total commit counts or issue numbers is insufficient. If a bug remains unpatched in one language but is fixed in another, it creates a risk for users. To evaluate the health of these libraries, we must analyze how individual fixes propagate between languages.

2.3 Research Contributions

Previous research on cross-ecosystem libraries has focused on: (1) dependency structures between package managers [6], and (2) repository-level activity metrics [9]. These studies show that these libraries are common, popular, and long-lasting. However, they treat the project as a single unit. Recent studies have begun analyzing source code to track vulnerabilities [14], but how general functional bugs are maintained across languages remains unknown. Our study fills this gap by shifting the focus from total repository activity to individual bug-fix commits. We analyze repositories containing both Java and Python to clarify how bug fixes propagate between these languages. The contributions of this research are as follows:

1. **Refining the Granularity of Analysis:** We explicitly visualize how bug fixes appear across languages at the implementation level, which cannot be captured by general project metrics.
2. **Analysis Based on Bug Nature:** Using Large Language Models (LLMs) and topic modeling (BERTopic [4]), we classify bugs from commit messages and evaluate whether certain types of bugs are more likely to be fixed across languages.
3. **New Evaluation Perspective:** Beyond "sustainability," we propose a framework to evaluate "fix consistency" between languages to ensure user safety.

In summary, this research evaluates cross-ecosystem libraries based on whether fix consistency is maintained between languages. We aim to provide insights into how multi-language development affects actual quality assurance at the implementation level.

3 Dataset Extraction

3.1 Data Source

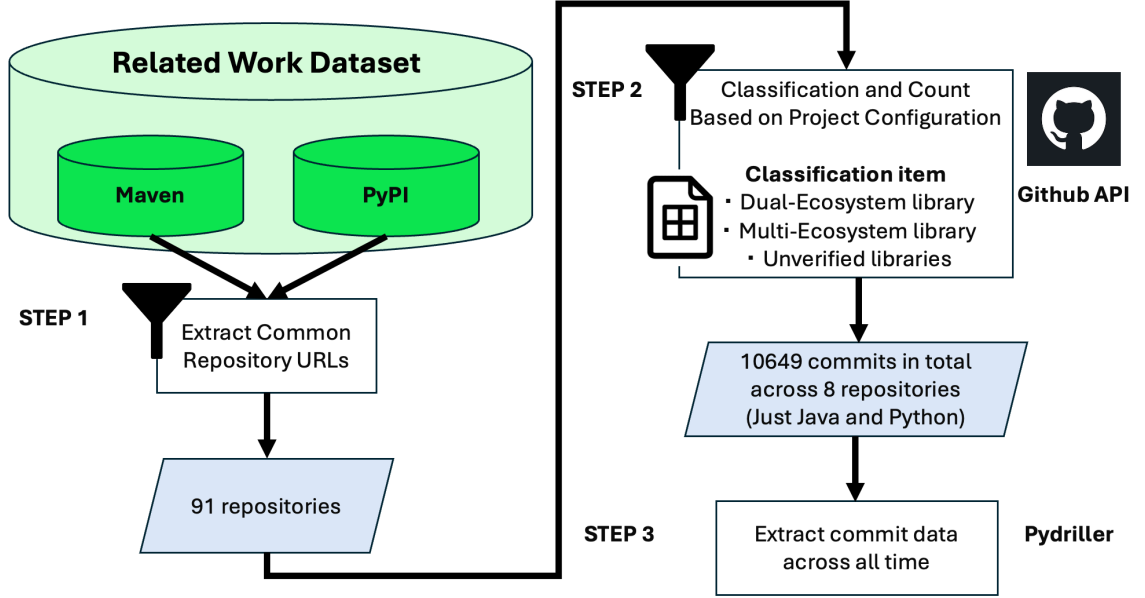


Figure 1: Data Collection and Selection Process

This section describes the dataset extraction process, including the identification of data sources and an overview of the data collection process. First, we explain the characteristics and selection criteria of the base dataset used in this study. Next, we specify the three-step procedure adopted to filter repositories and structure the commit data required for the analysis.

This study aims to elucidate the maintenance status of projects developed and distributed across the boundaries of disparate language ecosystems—specifically, traditional Java and the increasingly popular Python—through the analysis of commit messages. To identify target repositories and collect data, we employ the dataset published by Kannee et. al. [6]. The selected projects are verified as official, being hosted on both Maven Central (<https://central.sonatype.com/>) and PyPI (<https://pypi.org/>).

There are three primary reasons for selecting this dataset:

- **Suitability for Identifying Cross-Ecosystem Repositories**

The focus of this study is on projects that originate from a single GitHub repository while simultaneously releasing packages to both Maven Central (for Java) and PyPI (for Python).

Identifying such multi-ecosystem projects comprehensively using standard repository searches

is inherently difficult. However, since this dataset manages information by linking multiple package managers using GitHub URLs as a common key, it provides an optimal foundation for accurately and efficiently extracting the “cross-ecosystem repositories” required for this research.

- **Traceability for Detailed Analysis**

By using the repository URLs obtained from this dataset as a starting point, it is possible to retrieve the entire commit history for each project. This enables the construction of a dataset based on consistent criteria for investigating the ratio of bug-fix commits and differences in bug types between languages, which are addressed in subsequent Research Questions.

- **Novelty and Uniqueness**

The research that produced this dataset is one of the earliest studies on cross-ecosystem libraries. At present, there are no other publicly available datasets that manage repositories across multiple package managers. Therefore, using this dataset is essential to achieve the objectives of this study.

3.2 Overview of Data Collection Process

The data collection process follows three major stages to systematically identify and analyze target libraries, as illustrated in Fig 1. First, we extract common repository URLs shared between different ecosystems. Second, we verify the internal project structures to select libraries that are actively maintained in both languages. Finally, we extract and structure the complete commit history for detailed analysis. Each of these steps is described in detail in the following subsections.

3.2.1 Step 1: Extracting Common Repository URLs

In the initial stage of data collection, we extracted repositories deployed in both Java and Python ecosystems based on the aforementioned dataset. Specifically, we compared the registration Lists of Maven Central and PyPI to identify projects with matching repository URLs. This procedure allows for the extraction of projects that are managed and developed within a single repository despite being distributed across different language ecosystems. As a result of this inspection, 91 repositories with identical URLs were identified. These 91 repositories are considered to be maintained by the same development entities while possessing the characteristics of two distinct languages, making them appropriate subjects for the comparative analysis of this study.

3.2.2 Step 2: Verification of Project Structure and Selection

We verified whether the 91 projects extracted in Step 1 maintain both Java (Maven) and Python (PyPI) source code within the same repository. Specifically, we executed queries using GitHub API (Application Programming Interface) [3] to recursively retrieve the entire directory structure and performed the selection process according to the following procedures:

- **Repository Validation and Configuration Scanning**

First, we verified the accessibility of the target URLs and found that five repositories had been deleted or made private. For the remaining valid repositories, we examined the presence of language-specific build configuration files. We scanned for `pom.xml` in Java projects, and `setup.py`, `requirements.txt`, or `pyproject.toml` in Python projects to identify the current distribution format of each project.

- **Classification Based on Project Structure**

Based on the coexistence of these configuration files, we classified the projects into the following three categories (Table 1).

Table 1: Classification and Count Based on Project Configuration

Category	Definition / Classification Criteria	Count
Dual-Ecosystem library	Projects where only Java and Python configuration files are confirmed within the repository.	8
Multi-Ecosystem library	Projects where other language ecosystems, such as JavaScript (NPM) or Ruby, coexist alongside Java and Python.	16
Unverified libraries	Projects where only one language’s configuration file exists, or configuration files could not be confirmed.	62

The objective of this study is to quantitatively clarify the actual state of bug fix propagation in cross-ecosystem libraries deployed in both the Java and Python ecosystems. In projects where multiple programming languages coexist, dependencies between languages and development processes become more complex, which may introduce confounding variables into the analysis results. Therefore, by narrowing the scope to the 8 ”Dual-Ecosystem” projects, we aimed to limit the comparison subjects and enhance the validity of the interpretation of the analytical results.

3.2.3 Step 3: Commit History Extraction and Data Structuring

A potential risk in cross-ecosystem libraries is that a bug fixed in one language implementation may remain unpatched in the other ecosystem. To track the propagation and consistency of these bug fixes in detail, we extracted the complete commit history from the 8 repositories selected in Step 2 using PyDriller [13].

Specifically, we extracted and structured the following information from the components of each commit, as illustrated in Fig 2:

- **Commit Message (Intent): JaCM and PyCM**

This consists of text data (Title and Description) where maintainers describe the intent of the bug-fix. To classify the language-specific intent, we define:

- **JaCM (Java Commit Message):** Commits where the message contains Java-related keywords.
- **PyCM (Python Commit Message):** Commits where the message contains Python-related keywords.

These serve as the primary evidence for classifying technical factors (Type A–D) in RQ2.

- **Modified File Extension (Physical Change): JaFE and PyFE**

This includes path information for the files affected by the fix. We automatically identify file extensions to determine the actual scope of the change:

- **JaFE (Java File Extension):** Commits containing modifications to Java files (`.java`).
- **PyFE (Python File Extension):** Commits containing modifications to Python files (`.py`).

By linking the data of why the fix was made (commit message) and where the change occurred (files) at the single-commit level, we enabled a comprehensive quantitative and qualitative analysis of maintenance trends unique to multi-language projects.

```

hash,author,date,title,description,files_changed,num_files_changed
77330c37440cbb5d68c8f1025b6d9f3f2843122b,kun1996,2022-06-10 08:51:35+08:00,Add Windows support to CI (#487),"
test.yml
    Add windows-2019 to jobs.test.strategy.matrix.os

java_gateway.py
    In Windows, readline end with \r\n, use os.linesep instead of 1

java_gateway_test.py
    In windows stdout, \n will be replaced by \r\n
    In Java, System.out.println will output \r\n
    So, \r\n will be replaced by \r\r\n
    Use System.out.print to avoid platform differences

memory_leak_test.py
    I didn't think of a good way to solve it, maybe should open a new issue to trace it.

Resolves  #484

Co-authored-by: Hyukjin Kwon <gurwls223@gmail.com>,"test.yml|java_gateway.py|java_gateway_test.py|memory_leak_test.py,4

```

Figure 2: Commit History Example

4 Approach

This section describes the analysis methodology used to clarify bug-fixing activities in cross-ecosystem libraries. The analysis process consists of two main phases based on our Research Questions (RQs). First, in RQ1 analysis phase, we identify bug-fix commits from the project history. We map each fix to its target language using two metrics: file extensions and language-specific keywords. This allows us to evaluate the balance of maintenance activities in each project. Next, in RQ2 analysis phase, we perform topic modeling using BERTopic on projects with balanced maintenance. We extract the technical causes of bugs from commit messages. Based on a systematic taxonomy, we qualitatively and quantitatively clarify the differences in roles between Java and Python libraries, as well as their shared challenges.

4.1 RQ1: To what extent do bug-fix commits from dual-ecosystem libraries propagate to both code bases (Python and Java)?

In RQ1, we performed an analysis using the following steps to identify projects where substantial bug-fixing activities are continuously modified in both Java and Python files.

4.1.1 Extracting Bug-Fix Commit

First, we extracted commits related to bug fixes from the collected history based on the content of the commit messages. To improve classification precision, we implemented an extraction algorithm that categorizes keywords into "standalone terms" and "context-dependent terms."

- **Extraction Criteria:**

1. **Standalone terms** (Identifiable by keyword alone): `bug`, `fix`, `fixed`, `fixes`, `fixing`, `error`, `errors`
2. **Context-dependent terms** (Requiring an identifier): `defect`, `issue`, `fault`, `problem`, `regression`, `patch`, `close`, `resolve`, etc. Since these may be confused with feature changes when used alone, we confirmed their co-occurrence with an identifier (e.g., `#number`) which signifies a formal link to a bug-tracking system.

- **Exclusion Criteria:** Commits containing the following words were excluded as they often do not represent fixing activities: `release`, `version`, `merge`, `docs`, `refactor`, `update`.

4.1.2 Map Fixes to Programming Language

Next, we determined which language environment each bug-fix commit affected using two independent metrics:

1. Determination by File Extensions

We identified the target language by checking the extensions of the files modified in each commit. Specifically, if a commit includes changes to `.java` files, we categorize it as a Java fix. Similarly, if it includes changes to `.py` files, we categorize it as a Python fix. This allows us to confirm which language's source code was actually changed.

2. Determination by Commit Messages

We used regular expressions to check if the commit message contained language-specific keywords:

- **Java:** `java`, `maven`, `pom.xml`, `gradle`, `jar`, `jvm`
- **Python:** `python`, `pip`, `pypi`, `setup.py`, `requirements.txt`, `whl`, `pyc`

4.1.3 Classify Maintenance Balance

We calculated the percentage of Java-related fixes relative to all bug-fixing activities in each project to verify if both languages are maintained in parallel. To exclude projects with an extreme bias toward one language, we defined "Balanced Maintenance" as a state where the ratio of Java-related fixes falls between 15% and 85%. Based on this criteria, we classified the projects into four groups to evaluate the consistency between the two metrics (Table 2).

Table 2: Project Classification Based on Metrics

Category	Definition (Logic State)
Dual-Balanced Group (Commit Message-fileList consistent)	Projects where the ratio is balanced (15%–85%) in both extension and commit message metrics.
Message-Balanced Group (Commit FileList bias)	Projects balanced in commit message metrics but biased in extension metrics.
Extension-Balanced Group (Commit Message bias)	Projects balanced in extension metrics but biased in commit message metrics.
Unbalanced Group	Projects where fix activities are heavily biased toward one language in both metrics.

4.2 RQ2: Are there different to the kinds of bug fixes that propagate to the dual ecosystems?

In RQ2, we perform a qualitative and quantitative analysis of the bugs fixed in the "Balanced Maintenance" projects identified in RQ1. We use BERTopic, a method for extracting latent semantic structures from text, according to the following procedure.

4.2.1 Mining Commit Messages

To objectively classify the content of bug fixes, we applied topic modeling to the commit messages. To improve extraction accuracy, we cleaned the data based on the following criteria:

- **Cleaning Identifiers and Fixed Phrases:** We used regular expressions to remove Issue IDs (starting with #) and common words used in fixes (e.g., `fix`, `close`, `resolve`).
- **Exclusion of Numerical Information:** We removed version numbers and dates, as they represent administrative data rather than technical fix content.

- **Filtering Common Vocabulary:** In addition to standard English stop words, we used `CountVectorizer` to remove functional words that lack technical meaning.

4.2.2 Taxonomy of Bug-Fix Commit Messages

We examined the representative words and commit messages for each topic extracted by BERTopic. Based on the maintainers' intent expressed in the commit messages and the source of the bugs, we grouped them into the following four major technical categories:

- **Type A: Development Infrastructure & Build Configuration**

Definition: Issues related to build systems, dependency resolution, environment migration, and compilation settings.

Content: Errors in build scripts, library path settings, and compiler environment inconsistencies.

- **Type B: Static Rules & Coding Standards**

Definition: Rule violations detected by static analysis tools or compilers, and fixes to improve code formatting.

Content: Corrections of indentation/whitespace, warning suppression, introduction of type hints, and compliance with coding standards.

- **Type C: Runtime Interaction & Error Handling**

Definition: Management of dynamic behavior during execution, and improvements in error handling or logging.

Content: Communication control, multi-thread management, strengthening exception handling, and improving error log messages.

- **Type D: API Evolution & Long-term Maintenance**

Definition: Continuous adaptation to language specification changes and external library updates over time.

Content: Support for language version updates, API redefinition, and updates to copyright or license information.

Topics containing many specific system names or domain terms were treated separately as project-specific maintenance. By separating these, we aim to clearly capture more universal maintenance trends caused by the characteristics of the programming languages.

4.2.3 Analysis Perspectives

To evaluate the relationship between these technical categories and the languages (Java/Python), we established two analysis perspectives:

- **Perspective A (Cluster-wise Analysis):** We calculate the language distribution (Java-only / Python-only / Both) within each Type to analyze which language is mentioned more for specific technical factors.
- **Perspective B (Language-wise Analysis):** Within the Java-only or Python-only fix groups, we calculate the proportion of each Type to compare the different challenges faced by each language.

5 Results

5.1 RQ1: Distribution of Bug-Fix Activities by Language

This section describes the results of our investigation into the language-based distribution of bug-fix commits within the eight repositories where Java and Python coexist.

5.1.1 Classification of Projects Based on Metrics

First, we analyzed the occurrence rate of bug-fix commits identified through keyword searches across all commits in each repository. Table 3 shows the statistical results for the eight repositories.

Table 3: Bug-Fix Commit Rates by Repository

Repository name	Total Commits	Bug Fixes	Ratio
¹ py4j	798	218	27.3%
² appengine-pipelines	235	52	22.1%
³ geowave	2,149	403	18.8%
⁴ appengine-mapreduce	868	147	16.9%
⁵ appengine-gcs-client	292	48	16.4%
⁶ ahab	178	26	14.6%
⁷ HanLP	1,770	190	10.7%
⁸ ome-model	4,359	392	9.0%

The results show that the proportion of bug-fix activities ranges from 9.0% to 27.3% of the total development activity.

Next, we investigated the ratio of Java-related fixes within these bug-fix commits using two metrics: File Extensions (Ext) and Commit Messages (Msg). We identified projects in a "balanced state" (where Java-related fixes range from 15% to 85%). Our analysis confirmed that six out of the eight repositories fell within this range for both metrics. In contrast, for the remaining two repositories, Java-related modifications accounted for 98.7% to 100% of all bug fixes, indicating

¹<https://github.com/bartdag/py4j>

²<https://github.com/GoogleCloudPlatform/appengine-pipelines>

³<https://github.com/locationtech/geowave>

⁴<https://github.com/GoogleCloudPlatform/appengine-mapreduce>

⁵<https://github.com/GoogleCloudPlatform/appengine-gcs-client>

⁶<https://github.com/RENCI-NRIG/ahab>

⁷<https://github.com/hankcs/HanLP>

⁸<https://github.com/ome/ome-model>

that Python-side maintenance was extremely limited (Table 4).

Table 4: Maintenance Classification Based on Project Configuration

Category	Repository	Commit Count	Java Ratio (Ext)	Java Ratio (Msg)
Balanced Maintenance Group (6 projects, 1,047 total commits)	ome-model	392	66.1%	82.1%
	appengine-gcs-client	48	50.0%	42.9%
	appengine-pipelines	52	46.7%	60.0%
	appengine-mapreduce	147	40.7%	61.2%
	py4j	218	40.1%	38.2%
	HanLP	190	30.1%	80.0%
Biased Group (2 projects, 429 total commits)	geowave	403	98.7%	90.0%
	ahab	26	100.0%	0.0%

Based on these criteria, we selected the 1,047 commits from the "Balanced Maintenance Group"—where Java and Python are maintained in parallel—as the population for our detailed analysis.

5.1.2 Simultaneous Maintenance Rates by Repository

To evaluate how synchronized the bug-fixing activities are between Java and Python, we calculated two types of simultaneous activities for each repository:

- **Physical Synchronization (File level):** Modifying both Java and Python files in a single commit.
- **Logical Synchronization (Message level):** Mentioning both "Java" and "Python" in the commit message, regardless of which files were modified.

As shown in Table 5, these results quantitatively show that a single bug-fix commit rarely involves both languages simultaneously.

Table 5: Rates of Simultaneous Maintenance (Files vs. Messages)

Repository	Commit Count	Physical Synchronization (%)	Logical Synchronization (%)
py4j	218	24.31%	5.96%
appengine-pipelines	52	5.77%	0.00%
appengine-mapreduce	147	0.68%	1.36%
HanLP	190	0.53%	0.53%
ome-model	392	0.51%	0.26%
appengine-gcs-client	48	0.00%	0.00%

5.1.3 Overlap of Detection Metrics

Table 6 shows the detailed breakdown of how the detection metrics overlap within the Balanced Maintenance Group (1,047 commits).

Table 6: Detailed Intersection of Maintenance Metrics: Consistency between Commit Messages (CM) and Modified File Extensions (FE)

Metric Combination	Commit Count	Metric Combination	Commit Count
PyFE Only (Physical code fix only)	302	$\text{JaCM} \cap \text{PyCM} \cap \text{PyFE}$	8
JaFE Only (Physical code fix only)	213	$\text{JaCM} \cap \text{PyCM} \cap \text{JaFE} \cap \text{PyFE}$ (Full)	6
Neither (No language identified)	320	$\text{JaCM} \cap \text{JaFE} \cap \text{PyFE}$	5
$\text{PyCM} \cap \text{PyFE}$	58	$\text{JaCM} \cap \text{PyFE}$	5
$\text{JaCM} \cap \text{JaFE}$	50	PyCM Only (Intent mention only)	4
$\text{JaFE} \cap \text{PyFE}$ (Cross-language fix)	37	$\text{PyCM} \cap \text{JaFE}$	2
JaCM Only (Intent mention only)	25	$\text{JaCM} \cap \text{PyCM}$	1
$\text{PyCM} \cap \text{JaFE} \cap \text{PyFE}$	11	$\text{JaCM} \cap \text{PyCM} \cap \text{JaFE}$	0

Legend: Ja=Java, Py=Python, CM=Commit Message, FE=File Extension

5.2 RQ2 Relationship Between Technical Factors and Languages

This section presents the results of the analysis of technical factors (Type A–D) and their relationship with language categories, classified using the methodology described in Section 4.2. For this analysis, language categories are defined based on mentions within the commit messages as follows:

- **Java-only:** Mentions only Java-related vocabulary in the commit message.
- **Python-only:** Mentions only Python-related vocabulary in the commit message.
- **Both:** Mentions both Java and Python vocabulary.
- **Neither:** Does not explicitly mention vocabulary related to either language.

5.2.1 Language Mention Distribution per Technical Factor (Perspective A)

We investigated the tendency of language mentions within each technical Type. Table 7 shows the composition ratio of language categories for each Type.

The aggregation results revealed the following findings:

Table 7: Language Composition Ratio per Technical Factor (Perspective A)

Technical Type	Content Summary	Both (%)	Java-only (%)	Python-only (%)	Neither (%)
Type A	Build Infrastructure / Environment	0.00	15.85	0.61	83.54
Type B	Static Rules / Standards	0.00	2.63	2.63	94.74
Type C	Runtime / Error Handling	11.96	14.13	11.96	61.96
Type D	Evolution / Maintenance	1.54	1.54	55.38	41.54

- **Simultaneous Mentions of Multiple Languages:** In Type C (Runtime), the "Both" category reached 11.96%, which is significantly higher than in other Types.
- **Factors Dominant in Java:** In Type A (Infrastructure), Java-only mentions accounted for 15.85%, while Python-only mentions were extremely limited at 0.61%.
- **Factors Dominant in Python:** In Type D (Evolution), Python-only mentions accounted for 55.38%, showing the strongest bias toward a specific language among all Types.

5.2.2 Technical Factor Distribution per Language Category (Perspective B)

Next, we examined the proportion of each technical factor within the groups classified as Java-only and Python-only. The details are shown in Table 8.

Table 8: Proportion of Technical Factors within Each Language Category (Perspective B)

Language Category	Type A (%)	Type B (%)	Type C (%)	Type D (%)	Others (%)	Total (%)
Java-only	30.59	2.35	15.29	1.18	50.59	100.00
Python-only	1.33	2.67	14.67	48.00	33.33	100.00
Both	0.00	0.00	73.33	6.67	20.00	100.00

The characteristics of each language category are as follows:

- **Trends in Java-only:** Approximately 30% of Java-related fixes were concentrated in Type A (Infrastructure). This is remarkably high compared to the 1.33% seen in Python-only fixes.
- **Trends in Python-only:** For Python-related fixes, Type D (Evolution) showed the highest proportion at 48%.
- **Common Trends:** Regarding Type C (Runtime), both Java-only (15.29%) and Python-only (14.67%) appeared at almost identical rates, with a marginal difference of only 0.62%.

6 Discussion

In this section, I discuss the answers to the two research questions and some limitations of the study.

6.1 RQ1: Discussion on Maintenance Balance in Cross-Ecosystem Projects

In this section, we analyze the maintenance practices of projects where bug fixes are continuously performed in both Java and Python ecosystems. We combine quantitative analysis of commit data with manual inspection of repository structures.

6.1.1 Selection of Repositories for Comparative Analysis

We defined "Both Balanced" projects as those where the proportion of fixes for each language falls within the range of 15%–85%. This criterion does not only select projects with similar fix volumes. The unique feature of our method is analyzing the relationship between language mentions in commit messages and the actual files modified. If we include projects where only one language is maintained, we cannot observe this relationship effectively. Our manual check of excluded projects revealed that they used one language for core functions and the other as a wrapper. In such structures, bug fixes concentrate on the core language, leading to a strong bias. Therefore, our findings are limited to projects where both Java and Python handle substantial logic, implemented and maintained independently.

6.1.2 Classification of Bug Fix Trends Based on Implementation Structure

In cross-ecosystem libraries, the frequency of simultaneous fixes to Java and Python files depends heavily on the project's implementation structure. Based on our manual inspection, we identified two contrasting structures and their corresponding fix trends:

- **Type 1: Runtime Interdependency**

Structure: Java and Python interact during execution. If changes occur in interfaces or connection rules, the program will not function correctly unless both sides are updated (e.g., `py4j`).

Trend: To maintain consistency, fixes are often made to both languages simultaneously. Consequently, commits modifying both Java and Python files are observed at a relatively high frequency (24.3%).

- **Type 2: Independent Implementation**

Structure: The same functionality is implemented independently in each language. A fix in one language does not directly affect the execution of the other (e.g., HanLP).

Trend: Bug fixes are typically applied to one language first and then reflected in the other later. As a result, it is rare for a single commit to include fixes for both languages.

In particular, for projects with independent implementations, we confirmed that the proportion of simultaneous fixes to both Java and Python files within a single commit is very low. Crucially, this low rate does not necessarily mean that the consistency between languages is being neglected. Rather, it is likely a result of the codebases being separate, leading to fixes being applied at different times. Therefore, evaluating bug-fix consistency based solely on individual commits is insufficient. Since a bug fix made in one language may be reflected in the other with a certain time lag, analyzing the propagation of fixes across multiple commits remains a key challenge.

6.1.3 Intent Revealed by Mismatches Between File Fixes and Language Mentions

While the number of simultaneous fixes is small, there were some commits where the fixed files did not match the language mentioned in the commit message. For example, some commits modified only Java files but mentioned Python in the commit message ($JaFE \cap PyCM$), and vice versa. These commits show that maintainers are aware of the impact on the other language, not just the code they are fixing. In other words, even if the fix is limited to one language, the work is done to keep consistency with the other implementation. This can be seen as a state where intentions are shared through the commit message, even if both languages are not modified at the same time. This intentional consistency suggests that the lack of simultaneous code changes does not necessarily equate to a lack of maintenance awareness.

6.1.4 The Meaning of Commits Without Language Mentions

Further analysis revealed that in many commits, maintainers modified only Java or Python files without explicitly mentioning the language name in the commit message. Even in cross-ecosystem projects, maintainers rarely specify the language because the target is already obvious from the directory or file extensions.

Therefore, when a message mentions the other language, it reflects an uncommon maintainer behavior. In this study, we identified commits that mention both Java and Python in the commit message, even if the source code of both languages was not changed. These cases include two patterns:

- Commits where only one language's files were modified, but both languages were mentioned in the commit message.
- Commits where neither Java nor Python source code was modified (e.g., changes to documentation or configuration files), but the commit message refers to the relationship between both languages.

The appearance of a language name that is usually omitted suggests that the maintainer strongly recognizes the fix or issue as being relevant to both languages. Therefore, even without simultaneous file modifications, analyzing these mentions allows us to capture bug fixes whose intent or impact spans across both languages.

6.2 RQ2 Analysis Results: Qualitative Differences in Maintenance Between Ecosystems

The analysis for RQ2 revealed clear differences in the nature of bug fixes between Java and Python. This section integrates the findings from Perspective A and Perspective B to discuss the roles each language plays in cross-ecosystem development.

6.2.1 Type A: Build and Infrastructure Fixes Concentrated in Java

According to Perspective B, 30.59% of Java-related fixes were classified as Type A (Build Infrastructure / Environment). This is significantly higher than the 1.33% fixes found in Python, indicating that Java fixes are dominated by environment setup and build infrastructure. Since our target repositories were extracted from Maven Central, Maven is the primary build tool for the Java side. As a statically typed language, Java's dependency definitions and project configurations (`pom.xml`) in Maven tend to become complex. Consequently, constant adjustments are required to adapt to library updates and environment differences. The results suggest that maintenance in Java focuses on fine-tuning build settings and dependencies to ensure portability across different Operating Systems (OS) and Java versions, rather than frequent changes to functional logic. Thus, Java primarily serves to maintain the stable operational foundation of the project.

6.2.2 Type D: Evolution and Specification Fixes Concentrated in Python

In contrast, 48% of Python-related fixes were classified as Type D (Evolution / Maintenance), far exceeding the 1.18% in Java fixes. Python is frequently used in data science and machine learning, where it is combined with a vast number of external libraries. This leads to frequent reactive fixes due to updates in other libraries, API changes, or the evolution of Python's language specifications.

These fixes aim to continuously update the project's functions and adapt to changes in the external environment. This suggests that maintenance on the Python side focuses on functional evolution.

6.2.3 Type C: Shared Responsibility in Runtime Error Fixes

Type C (Runtime / Error Handling) appeared at almost identical rates in Java (15.29%) and Python (14.67%), with a difference of only 0.62%. Furthermore, this type showed a high percentage of "Both" mentions in commit messages. Bugs related to runtime exception handling and control flow often occur at the interface or coordination points between the two languages rather than within a single language. Therefore, for these types of fixes, it is inferred that maintainers work while simultaneously considering the behavior of both languages. While the roles for build management and functional updates are divided, both languages are closely linked when it comes to ensuring runtime safety. However, it remains to be seen if this role separation holds true in different structural patterns, such as wrapper-based projects, or other language pairs like Python and C++. This suggests a need for broader investigation to generalize these maintenance roles.

6.2.4 Limitations

While we found that Java handles Type A and Python handles Type D, some caution is required. As shown in Perspective A, more than half of the commits did not mention a specific language (Neither), meaning explicit data containing language mentions is limited. Additionally, with a small number of language-specific commits, a temporary large-scale fix in a specific project may have biased the overall statistics. Therefore, the role division identified here should be interpreted as a strong trend observed in our specific subjects and period, rather than a universal rule of cross-ecosystem development. Additionally, our classification relied on commit messages and file extensions. While effective for high-level categorization, this approach does not capture the fine-grained semantic changes within the code. Incorporating source code diff analysis would provide a deeper understanding of how logic is synchronized across ecosystems.

Generalizing these findings will require future analysis with a larger variety of projects and different development phases.

7 Conclusion and Future work

7.1 Research Summary

This study analyzed bug-fix commits in cross-ecosystem libraries where both Java and Python ecosystems are continuously maintained. Our analysis combined quantitative methods based on bug fix frequency and scope with qualitative methods based on the classification of bug-fix content.

7.1.1 Separation of Fix Units in Maintenance Activities (RQ1)

The results of RQ1 show that even when Java and Python are fixed with similar frequency, the rate of simultaneous fixes within a single commit depends heavily on the project’s implementation structure. In projects with runtime interdependencies, this rate reached 24.3%, suggesting that simultaneous updates are appropriately performed where they are essential to avoid critical errors. In contrast, projects with independent implementations showed an extremely low rate (average 1.574%), indicating that fixes are typically applied in separate commits. However, even when physical simultaneous modifications do not occur, our study identified an invisible form of propagation through language mentions in commit messages. In cross-ecosystem development, mentioning a language name is often redundant and typically omitted when the target is obvious. Therefore, cases where a maintainer explicitly mentions the other language without modifying its code suggest that they are conscious of the consistency between both languages. Consequently, while direct propagation via simultaneous file changes is limited within a single commit, the presence of these mentions indicates that maintainers recognize certain fixes as being common to, or affecting, both codebases. While the time lag between physical code changes requires further analysis, the appearance of the other language’s name in messages is a significant indicator that the fix is not perceived as an isolated issue. We conclude that when evaluating the maintenance of multi-language projects, synchronization at the developer’s consciousness level functions as a vital element in maintaining project integrity—a factor that cannot be captured solely by observing simultaneous file modifications within a single commit.

7.1.2 Differences in Maintenance Content Between Ecosystems (RQ2)

The analysis for RQ2 confirmed clear differences in the distribution of bug-fix content between Java and Python:

- **Java Maintenance:** 30.59% of Java-related fixes were categorized as foundational tasks, such as build settings and dependency management (Type A). This shows that a significant

amount of Java maintenance is dedicated to stabilizing Maven-based build configurations and execution environments.

- **Python Maintenance:** 48% of Python-related fixes were categorized as adaptation to specification changes and long-term maintenance (Type D). This indicates that Python fixes are more frequent in areas such as responding to updates in external libraries.
- **Common Fixes for Both Languages:** For fixes related to runtime control and exception handling (Type C), the frequency was nearly identical for both Java and Python. Furthermore, 73.33% of "Both" mentions (i.e., commits referring to both Java and Python in their messages) were concentrated in this type. This shows that responding to runtime errors is treated as a shared concern, regardless of the language.

In conclusion, these results clarify a division of roles in cross-ecosystem libraries: Java primarily handles the stability of the build and execution environment, while Python handles adaptation to specification changes. Although each language has different maintenance targets, they maintain the overall quality of the library by addressing runtime issues as a common problem.

7.2 Future Work

This study is a preliminary investigation into bug-fix activities in cross-ecosystem libraries. Based on our findings, we identify four key areas for future research.

7.2.1 Future Work 1: Improving Analysis Methods: Source Code Diff Analysis

In this study, we classified fixes primarily using commit messages and file extensions. However, this method might miss small logic changes or type definition updates not explicitly mentioned in the commit messages. Future work should introduce direct analysis of source code diffs. This will allow us to confirm exactly how a bug fix in one language is reflected in another at the code level. This approach will serve as a foundation for the other research goals listed below.

7.2.2 Future Work 2: Analyzing Bug Impacts in Wrapper Structures

Our research focused on balanced projects where Java and Python are fixed with similar frequency. However, in wrapper-structured projects—where one language handles core processing and the other calls those functions—the dependencies are more explicit. We plan to investigate how a specific bug in the core implementation appears as a bug in the wrapper implementation. By

tracking error names and exception types in diffs and commit messages, we can clarify the chain of bug impacts across languages, which is harder to see in balanced projects.

7.2.3 Future Work 3: Quantitative Analysis of Fix Timing and Delays

The insights gained from this study show that while simultaneous fixes to both Java and Python files in a single commit are rare, there are cases where the commit message mentions a language that was not actually fixed. This suggests that consistency between languages is maintained through multiple fixes over time, rather than within a single commit. In future work, we will quantitatively analyze the time lag between a bug fix in one language and its reflection in the other. We plan to use common issue IDs and code diffs as primary evidence for this analysis. This approach will allow us to measure the time lag of "recognized propagation" found in messages, as well as identify "hidden propagation" that can only be detected through code similarity (diffs). By doing so, we aim to clarify the cross-ecosystem bug-fix propagation process using objective data that does not depend on the maintainer's writing habits.

7.2.4 Future Work 4: Expanding the Dataset and Target Languages

To confirm whether our findings—Java handling build infrastructure and Python handling functional evolution—are universal, we need to analyze more projects. Future studies will increase the number of repositories to reduce the risk of results being biased by specific projects. Furthermore, we will expand our scope to include other combinations, such as Python with system languages like C++ or Rust, as well as projects involving three or more languages. This will help us understand how different language types and combinations influence the division of maintenance tasks.

Acknowledgement

First and foremost, I would like to express my sincere gratitude to everyone who contributed to the investigation and writing of this thesis.

First of all, I would like to express my deepest gratitude to my supervisor, Professor Raula Gaikovina Kula, for his invaluable guidance, constant encouragement, and insightful feedback throughout this research.

I am also sincerely grateful to ssociate Professor Olivier Nourry for his constructive discussions and guidance regarding the research direction during my mid-term presentation.

I would like to extend my sincere thanks to Professor Yoshiki Higo, Associate Professor Makoto Matsushita, Assistant Professor Shinsuke Matsumoto, and Professor Shinji Kusumoto for their vital feedback on the direction and validity of this study.

I want to express my sincere thanks to Warren Gerald Polandra, Nigar Alizada, and Inase Kondo for their dedicated efforts in refining my thesis.

I am thankful to Chihiro Yoshida for her thorough review and helpful feedback on my research direction and presentation materials.

I am also thankful to Ms. Mizuho Karube and Ms. Takara Miyazaki for their administrative support and for maintaining a comfortable research environment, which allowed me to focus entirely on my work.

Finally, I wish to express my heartfelt appreciation to all the members of the Higo Laboratory at the Graduate School of Information Science and Technology, Osaka University. I am profoundly grateful for their unwavering support and encouragement; they have always been there for me, offering their help selflessly and without expectation. This work would not have been possible without their presence.

References

- [1] Eleni Constantinou, Alexandre Decan, and Tom Mens. Breaking the borders: an investigation of cross-ecosystem software packages, 2018.
- [2] Adam Saphiro Dewayanto, Yusuf Sulisty Nugroho, and Raula Gaikovina Kula. Visualizing the synchronicity of fixed issues across diverse ecosystems. In *2024 International Conference on Smart Computing, IoT and Machine Learning (SIML)*, pages 124–129, 2024.
- [3] GitHub, Inc. Github rest api documentation. <https://docs.github.com/en/rest>, 2022. Accessed: 2026-02-02.
- [4] Maarten Grootendorst. Bertopic: Neural topic modeling with a class-based tf-idf procedure. *arXiv preprint arXiv:2203.05794*, 2022.
- [5] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. Characterizing usages, updates and risks of third-party libraries in java projects. *Empirical Software Engineering*, 27(4):90, Apr 2022.
- [6] Kanchanok Kannee, Raula Gaikovina Kula, Supatsara Wattanakriengkrai, and Kenichi Matsumoto. Intertwining communities: Exploring libraries that cross software ecosystems. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 518–522, 2023.
- [7] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, Feb 2018.
- [8] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. Selecting third-party libraries: the practitioners ’ perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 245–256, New York, NY, USA, 2020. Association for Computing Machinery.
- [9] Benson MIROU Kenichi MATSUMOTO Raula GAIKOVINA KULA Lenz NERIT, Youmei FAN. Should maintainers deploy to cross-ecosystems? an analysis of packages from pypi and npm. *IEICE TRANSACTIONS on Information*, (0):0–0, January.

- [10] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. *Understanding the Response to Open-Source Dependency Abandonment in the npm Ecosystem*, page 2355–2367. IEEE Press, 2025.
- [11] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 84–94, 2017.
- [12] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E. Santosa, Asankhaya Sharma, and David Lo. Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26(4):59, Apr 2021.
- [13] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.
- [14] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1046–1058, 2023.
- [15] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the diversity of software package popularity metrics: An empirical study of npm, 2019.