MiG.4: A Curated Dataset of Library Migrations in Java and Python

Matheus Barbosa*
Pedro Baptista*
matheus.grandinetti@dcc.ufmg.br
baptistapedro@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

João Eduardo Montandon
joao@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Abstract

Software library and framework migrations are common, yet challenging and time-consuming tasks in software evolution. The development and rigorous evaluation of automated migration tools, particularly those based on Large Language Models (LLMs), require high-quality, real-world datasets that are free from noise. We propose MiG.4, a manually curated dataset of real-world code migrations performed between four popular Java and Python libraries. Currently, the dataset comprises 800 instances of isolated, developer-performed migrations, focusing on four highly relevant scenarios: $JUnit \Leftrightarrow TestNG$, $Mockito \Leftrightarrow EasyMock$, $Urllib \Rightarrow Requests$, and $Boto \Rightarrow Boto3$. Each migration record is structured with eight fields, including the code snippets before and after the change, and a classification that distinguishes between Simple and Complex Migrations. Our dataset serves as a solid ground truth, facilitating future empirical studies and development of novel solutions for automated third-components migration.

Dataset Package. The dataset is publicly available at: https://doi.org/10.5281/zenodo.17665212

ACM Reference Format:
Matheus Barbosa, Pedro Baptista, and João Eduardo Montandon. 2026. MiG.4: A Curated Dataset of Library Migrations in Java and Python. In *2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering (FORGE '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793655.3793711

1 Introduction

The raise of third-party libraries and frameworks has changed the way software is developed [6, 8]. Nowadays, developers extensively rely on pre-built functionalities provided by these components to ship software faster and with higher quality [9–11]. For a given task, developers have at their disposal a wide range of libraries and frameworks to choose from, each with its own set of features, including API structure, community guidelines, API documentation, and licensing terms [7, 8]. While React, Vue, and Angular are popular choices for building user interfaces and managing applications, developers can implement the backend of their server-side applications using either Flask, Django, or FastAPI.

These components might be changed as the software project evolves. For example, GitHub's engineering team upgraded their codebase from Rails 3.2 to Rails 5.2. The migration process took over a year, allowing them to fix technical debts and use new features available in the newer version of the framework.[1] Likewise, many open source software maintainers who opted for Python's unittest library to write their test cases decided to migrate their test suite to pytest, a modern and feature-rich testing library [2, 3].

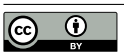
It turns out that these migrations require a high cognitive load since developers manually migrate their code to the new library or framework. Besides demanding expertise in both legacy and target APIs perform the migration, they must know how to correctly perform this transformation in the codebase they are working on. This task is error-prone and may introduce new bugs in the migrated source code. In recent years, Large Language Models (LLMs) have shown promising capabilities in various code migration tasks, including updating library versions, and replacing one library for another [1, 2, 6]. Unfortunately, the dataset used in these studies present some restrictions, such as a limited number of migration scenarios, the presence of other changes besides the migration code, and a lack of variety in the migration scenarios.

Our Proposal. To address this gap, MiG.4 is introduced as a manually curated dataset of real-world code migrations extracted from prior work [4]. The dataset comprises 800 isolated and verified migration instances, each representing an authentic developer-performed modification. MiG.4 encompasses migrations involving widely adopted libraries across two major programming ecosystems—Java (EasyMock, Mockito, JUnit, and TestNG) and Python (Urllib, Requests, Boto, and Boto3).

Beyond its scale, MiG.4 introduces three key innovations that distinguish it from existing resources:

- **Rigorous curation process**: all migration instances were manually inspected, validated, and refined to ensure correctness and eliminate false positives, resulting in a high-confidence dataset.
- **Cross-language diversity**: the dataset integrates migration scenarios from both Java and Python ecosystems, allowing comparative studies across languages and runtime environments.

*Both authors contributed equally to this research.



FORGE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2477-0/2026/04
https://doi.org/10.1145/3793655.3793711

[1] https://github.blog/engineering/infrastructure/upgrading-github-from-rails-3-2-to-5-2/

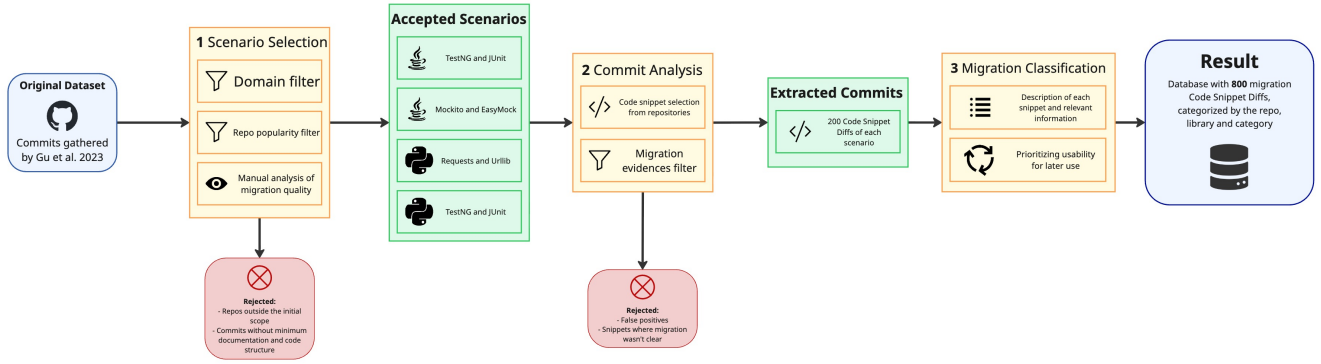


Figure 1: Overview of the data collection methodology.

- **Varied migration types**: varied migration types — MiG.4 captures heterogeneous migration patterns, including inter-library transitions (e.g., Mockito ⇔ EasyMock) and intra-library version upgrades (e.g., Boto ⇒ Boto3), thus reflecting the full spectrum of migration complexity encountered in modern software evolution.

Regarding its utility, MiG.4 is designed to serve as a high-quality benchmark for training and evaluating Large Language Models (LLMs) and other AI-driven approaches in the context of software evolution. By providing a clean, "ground truth" source of human-written migrations, it enables researchers to perform fine-grained comparative studies and develop more accurate automated refactoring tools. Ultimately, MiG.4 acts as a vital auxiliary resource in software engineering research, helping to bridge the gap between manual migration efforts and autonomous coding assistants.

Beyond its role in AI research, MiG.4 serves as a valuable asset for broader software engineering endeavors. It can be used to mine common refactoring patterns, providing a catalog of real-world API mapping challenges. Furthermore, the dataset acts as a robust testbed for static analysis tools to validate automated quick-fixes. From an educational perspective, MiG.4 offers a curated repository of practical evolution cases, allowing students and practitioners to study the complexities of library dependencies and technical debt management in large-scale projects.

Dataset Availability. The MiG.4 dataset is publicly available at https://doi.org/10.5281/zenodo.17665212.

2 Data Collection

We developed a four-step data collection pipeline to build the MiG.4 dataset, as illustrated in Figure 1. Each stage of the process is described below.

The Original Dataset. Our dataset builds upon an existing collection of library migration instances, originally compiled by Gu et al. [4]. The original dataset contains an extensive corpus of 33,667 commits flagged as library migrations in different third-party components and programming languages. However, the dataset provides only commit-level information—e.g., it does not contain the code snippets or diffs that represent the changes that occurred during the migration—which limits its utility for fine-grained migration analysis. We propose to improve this dataset by extracting the migrations effectively performed at source code level.

Scenario Selection. Two authors independently reviewed the original dataset to identify migration scenarios that is worth improving. For this selection, the authors analyzed the migration scenarios present in the dataset, and filtered those according to the following criteria.

- **Library popularity**: we considered migrations involving widely adopted and well-established libraries;
- **Migration type**: we selected migration pairs to ensure both inter-library transitions and intra-library version upgrades.
- **Cross-language support**: we included migration scenarios performed in two programming languages: Java and Python.

Table 1 summarizes the migration pairs selected after this inspection. In total, we selected four migration scenarios: $TestNG \Leftrightarrow JUnit$ and $Mockito \Leftrightarrow EasyMock$ for Java, and $Urllib \Rightarrow Requests$ and $Boto \Rightarrow Boto3$ for Python. Three of these pairs represent inter-library migrations, while one corresponds to an intra-library upgrade. Furthermore, we included bi-directional migrations for two scenarios, represented by the double arrows (⇔). These pairs contain migration examples performed both forward (A ⇒ B) and backward (B ⇒ A), allowing researchers to study differences in migration complexity and patterns in both directions. In our view, this selection provides a balanced yet diverse set of migration scenarios, similar to those investigated in the literature [1, 3, 5, 6].

Table 1: Selected migration pairs for dataset curation.

Programming Language	Migration Pair	Type	Commits
Java	TestNG ⇔ JUnit	Inter	20
	Mockito ⇔ EasyMock	Inter	10
Python	Urllib ⇒ Requests	Inter	82
	Boto ⇒ Boto3	Intra	68

Commit Analysis. Once the migration scenarios were selected, we proceeded to extract the actual source code changes corresponding to each migration instance. For each commit listed in the original dataset corresponding to one of the selected scenarios, two authors read the commit diff to locate the code segments that effectively performed the migration between the legacy and target libraries. The authors selected a code diff as a migration instance if the *modification explicitly included the replacement of API calls, data structures, or other constructs associated with the libraries involved in the migration.* During this step, the authors isolated each migration instance from unrelated code changes performed in the same commit. Finally, commits considered false positives were discarded.

For each scenario, the authors performed the classification until a limit of 200 migration instances was reached. This threshold ensured a balanced representation among the selected migration pairs, resulting in a total of 800 migration instances.

Migration Classification. In this final step, each migration instance was classified according to its level of complexity. Specifically, migration instances were categorized as either *Simple* or *Complex* migrations. A simple migration corresponds to cases involving minimal syntactic changes, where the overall code structure remains largely unchanged during migration. For example, a unit test that replaced a function call from the legacy library with one from the target library—without modifying its parameters or the surrounding code—would be considered a simple migration. In other words, migration instances were classified as simple if the changes made are direct and require no other additions or removals to the code other than the signature change. On the other hand, complex migrations represent larger transformations, where the structural aspect of the modified code is significantly modified. These show significant differences in the libraries that made necessary additions or removals in the code structure, such as assigning new variables, calling new functions, or the removal of such aspects.

3 Dataset Structure

Table 2 provides an overview of the MiG.4 dataset. As mentioned above, the dataset encompasses 800 migration instances, evenly distributed across four migration scenarios. As we can see, 479 (59.9%) of the migrations were classified as *Simple*, while 321 (40.1%) were labeled as *Complex*. Most migration scenarios exhibit similar distributions between simple and complex migrations. The *Mockito ⇔ EasyMock* scenario stands out with a greater number of complex migrations (108) than simple ones (92).

Table 2: Overview of the MiG.4 dataset.

Migration Scenario	# of Migration Instances		Total
	Simple	Complex	
TestNG ⇔ JUnit	140	60	200
Mockito ⇔ EasyMock	92	108	200
Urllib ⇒ Requests	129	71	200
Boto ⇒ Boto3	118	82	200
Total	479	321	800

Currently, the dataset is organized as a structured collection of records, stored in CSV format. Each record represents a single, isolated migration instance, and includes eight fields describing the migration metadata and the associated code snippets before and after the migration.

- **repo_name:** The name of the GitHub repository where the migration occurred.
- **commit_code:** The commit hash where the migration was performed.
- **file_name:** Name of the file affected by the migration.
- **type:** The classification of the migration as either *Simple* or *Complex*.
- **legacy_lib:** The name of the original library or framework being migrated from.
- **target_lib:** The name of the target library or framework being migrated to.
- **code_before:** The code snippet containing the usage of the legacy library before the migration.
- **code_after:** The code snippet containing the usage of the target library after the migration.

4 Migration Examples

Figure 2 illustrates two migration instances from the MiG.4 dataset. As previously stated, these migrations were isolated from other changes to include only the code changed to perform the transition between legacy and target libraries. We explain each example in detail below.

4.1 Simple Migration

Figure 2a presents a simple migration on *boto ⇒ boto3*. This migration was conducted in 2016 in the Cloud Tools Stacker project.[2] The whole commit contains 51 changes performed in 9 files, totaling 117 lines changed.

As previously stated, a simple migration is characterized by straightforward changes to the source code, such as replacing a function call or modifying its parameters, without modifying the overall code structure. This specific instance is characterized by a direct one-to-one mapping of changes: the single call to connect using `cloudformation` is replaced by a two-step initialization process. This new process involves creating a Boto3 session followed by retrieving the client via the built-in client method. Because the migration replaces only library-specific initialization elements for improved clarity and adherence to the current AWS SDK best practices, as described in the migration commit, this transition was categorized as a simple migration.

4.2 Complex Migration

Figure 2b demonstrates a complex migration involving the transition from *EasyMock ⇒ Mockito*. This migration was conducted in 2013 in the Spring Batch project.[3] The whole commit contains 94 changes performed in 78 files, totaling 1,279 lines changed.

Unlike simple migrations—which often consist of one-to-one function call substitutions—this case shows a fundamental shift in the testing strategy—moving from EasyMock's imperative "Record-Replay" model to Mockito's declarative "Arrange-Act-Assert" model.

[2] https://github.com/cloudtools/stacker/commit/f7a250072d1d8af6352f49044ec5570ac47378f2
[3] https://github.com/spring-projects/spring-batch/commit/7e1e66d677d8cdd9fbf1c9e07e5b307a249d9e05

```
83        @property
84        def cloudformation(self):
85            if not hasattr(self, "_cloudformation"):
86  -             self._cloudformation =
    cloudformation.connect_to_region(
87  -                 self.region)
88            return self._cloudformation
```

```
93        @property
94        def cloudformation(self):
95            if not hasattr(self, "_cloudformation"):
96  +             session = boto3.Session(profile_name=self.profile,
    region_name=self.region)
97  +             self._cloudformation =
    session.client('cloudformation')
98            return self._cloudformation
```

(a) A simple migration example from *Boto* ⇒ *Boto3*.

```
75        @Test
76        public void testGetJobExecution() throws Exception {
77  -
    expect(jobExecutionDao.getJobExecution(123L)).andReturn(jobExecution);
78  -         expect(jobInstanceDao.getJobInstance(jobExecution)).andReturn(
79                  jobInstance);
80            stepExecutionDao.addStepExecutions(jobExecution);
81  -         expectLastCall();
82  -         replay(jobExecutionDao, jobInstanceDao, stepExecutionDao);
83            jobExplorer.getJobExecution(123L);
84  -         verify(jobExecutionDao, jobInstanceDao, stepExecutionDao);
85        }
```

```
73        @Test
74        public void testGetJobExecution() throws Exception {
75  +
    when(jobExecutionDao.getJobExecution(123L)).thenReturn(jobExecution);
76  +         when(jobInstanceDao.getJobInstance(jobExecution)).thenReturn(
77                  jobInstance);
78            stepExecutionDao.addStepExecutions(jobExecution);
79            jobExplorer.getJobExecution(123L);
80        }
```

(b) A complex migration example from *EasyMock* ⇒ *Mockito*.

Figure 2: Examples of simple and complex migrations from the MiG.4 dataset.

To perform this transition, the maintainer removed the state-management methods such as `replay()` and `expectLastCall()`. Additionally, the developer reestructured the mock behavior, converting `expect().andReturn()` to the `when().thenReturn()` chain. These changes represent a modification of the test's control flow and state logic, fiting our criteria for complex migrations.

5 Related Work

Library migration involves replacing one library with another in a software project [4, 6]. Barbosa and Hora [3] analyzed the reasons for migrating from *unittest* to *pytest* in open-source Python projects, showing that developers take into account new features, easier syntax, and flexibility when deciding to migrate. Alves and Hora [2] extended this work to provide a curated dataset of *unittest* to *pytest* source code migrations. Islam et al. [6] mapped the changes involving library migration in 311 Python projects, and leveraged a taxonomy with the strutural changes needed to perform the migrations. Besides changing the functions, developers adopt different program elements—e.g., use decorators, include exception handling, etc—when conducting library migrations.

Gu et al. [4] conducted a comparative study about library migration in three popular ecosystems—Java/Maven, JavaScript/npm, and Python/PyPI—to understand in which context developers are more likely to perform library migrations. The authors showed that different ecosystems share similar motivations to replace one library with another, such as old library presenting issues, target library offering new features, etc. Similar to Alves and Hora [2], we meticulously analyzed and augmented the dataset presented in Gu et al. [4] to include the source diffs with the migration effectively performed by the developers, ensuring a more detailed and reliable representation of migration scenarios. These additions encompass a wider variety of libraries, frameworks, and programming languages, facilitating a more robust foundation for automated migration research.

6 Conclusion

In this work, we presented MiG.4, a manually curated dataset of real-world code migrations involving Java and Python libraries. At its current state, the dataset contains 800 isolated migration instances from four relevant migration scenarios, each classified according to its complexity level. We believe that MiG.4 serves as a solid dataset to support further research on automated solutions for library migration tasks at source code level.

Future Work. At short term, we plan to extend the collection of migration instances with new instances and scenarios. Another important direction is to improve the characterization of the dataset by analyzing the types of modifications performed during migrations. With this detailed taxonomy, we can conduct a more precise evaluation of migration techniques with respect to these patterns. Finally, we intend to use the dataset in empirical studies to assess the capabilities of Large Language Models (LLMs) in automating code migration tasks. Finally, we intend to leverage the dataset in future empirical studies to evaluate the capabilities of Large Language Models (LLMs) in assisting code migration tasks.

Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. It was also supported by grants from CNPq (403304/2025-3) and by FAPEMIG (APQ-02419-23).

References

[1] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 2024. Automatic Library Migration Using Large Language Models: First Results. In *18th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–7.

[2] Altino Alves and Andre Hora. 2025. TestMigrationsInPy: A Dataset of Test Migrations from Unittest to Pytest. In *Mining Software Repositories (MSR): Data and Tools Showcase Track*. 1–5.

[3] Livia Barbosa and Andre Hora. 2022. How and Why Developers Migrate Python Tests. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 538–548.

[4] Haiqiao Gu, Hao He, and Minghui Zhou. 2023. Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 627–638.

[5] Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, and Xin Peng. 2021. REPFINDER: Finding Replacements for Missing APIs in Library Update. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 266–278.

[6] Mohayeminul Islam, Ajay Kumar Jha, Ildar Akhmetov, and Sarah Nadi. 2024. Characterizing Python Library Migrations. In *ACM International Conference on the Foundations of Software Engineering (FSE)*, Vol. 1. 1–23.

[7] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A Systematic Review of API Evolution Literature. *Comput. Surveys* 54, 8 (Oct. 2021), 171:1–171:36.

[8] Maxime Lamothe and Weiyi Shang. 2018. Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android. In *15th International Conference on Mining Software Repositories (MSR)*. 503–514.

[9] Guilherme Miranda, João Eduardo Montandon, and Marco Tulio Valente. 2022. TechSpaces: Identifying and Clustering Popular Programming Technologies. In *16th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*. 60–67.

[10] João Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying Experts in Software Libraries and Frameworks Among GitHub Users. In *16th International Conference on Mining Software Repositories (MSR)*. 276–287.

[11] João Eduardo Montandon, Cristiano Politowski, Luciana Lourdes Silva, Marco Tulio Valente, Fabio Petrillo, and Yann Gaël Guéhéneuc. 2021. What Skills Do IT Companies Look for in New Developers? A Study with Stack Overflow Jobs. *Information and Software Technology* 129 (2021), 1–6.