PyMigBench: A Benchmark for Python Library Migration

Mohayeminul Islam
University of Alberta
Edmonton, Canada
mohayemin@ualberta.ca

Ajay Kumar Jha
North Dakota State University
Fargo, USA
ajay.jha.1@ndsu.edu

Sarah Nadi
University of Alberta
Edmonton, Canada
nadi@ualberta.ca

Ildar Akhmetov
University of Alberta
Edmonton, Canada
ildar@ualberta.ca

***Abstract*—Developers heavily rely on Application Programming Interfaces (APIs) from libraries to build their projects. However, libraries might become obsolete, or new libraries with better APIs might become available. In such cases, developers replace the used libraries with alternative libraries, a process known as *library migration*. Since manually migrating between libraries is tedious and error prone, there has been a lot of effort towards automated library migration. However, most of the current research on automated library migration focuses on Java libraries, and even more so on version migrations of the same library. Despite the increasing popularity of Python, limited research has investigated migration between Python libraries. To provide the necessary data for advancing the development of Python library migration tools, this paper contributes PyMigBench, a benchmark of real Python library migrations. PyMigBench contains 59 analogous library pairs and 75 real migrations with migration-related code changes in 161 Python files across 57 client repositories.**

***Index Terms*—Python, library migration, migration-related code changes, benchmark**

I. Introduction

Developers heavily rely on the Application Programming Interfaces (APIs) offered by libraries to build software. However, the libraries and APIs an application depends on may become obsolete over time [?], [?], [?]. Libraries may also negatively impact the applications that use them due to critical vulnerabilities or bugs present in the libraries [?], [?]. Developers may also find new, better-performing, or easier-to-use libraries [?], [?], [?]. In all such cases, developers replace the currently used library with an alternate one, a process referred to as *library migration*.

The library migration process typically involves finding an alternative library with the required functionality (*library mapping*), finding the replacements for each usage of an API of the old library (*API mapping*), and updating all existing code that used the old library's API to now use the new library's API while preserving the software's behavior (*client code transformation*). This is a tedious and error-prone task that developers often dread [?]. Techniques that automate this entire migration process can save developers time and effort.

Researchers have attempted to automate the different activities of the migration process, such as identifying and recommending alternative (a.k.a *analogous*) libraries [?], [?], [?], [?], [?], [?] and APIs [?], [?], [?], [?], [?], [?], [?] and transforming client code [?], [?], [?], [?]. While it is evident that library migration is an active research area with lots of advancements, there are still gaps in the literature. First, most of these techniques focus on Java [?], [?], [?], [?], [?], [?], [?], [?], [?], [?]. It is not clear whether these techniques can be applied to a different programming language that is not statically typed, such as Python [?], [?]. Second, most of the client code transformation techniques focus on *version migration* (i.e., migration between different versions of the same library) [?], [?], [?]. Third, researchers have evaluated most of the migration techniques by migrating code for a few arbitrarily selected library pairs with no standard library migration benchmarks to facilitate evaluation and comparison.

To address these gaps, we mine open-source Python repositories and create a benchmark of Python library migrations, PyMigBench. PyMigBench has a total of 59 manually confirmed analogous library pairs and 157 migrations. From these, 75 migrations between 34 library pairs have associated code changes in 57 different client repositories. These code changes span 161 Python files with 375 code change segments. Researchers can leverage PyMigBench for building Python library migration tools. Specifically, researchers can study the real migrations in PyMigBench to understand the types of code changes that need to be supported for Python library migration and can use PyMigBench as a ground truth for evaluating and comparing migration approaches. The dataset and associated tools are available in our repository[1].

II. Constructing PyMigBench

We use automated (white boxes) and manual (grey boxes) steps shown in Figure 1 to construct PyMigBench. We first fetch and clone Python repositories (Step 1). We then iterate through the commits in these repositories to identify commits that potentially contain library migrations, *candidate migration commits* (Step 2). We analyze the candidate migration commits and collect candidate migrations (Step 3). After this, we confirm analogous library pairs between which we observe candidate migrations (Step 4), then we confirm the migrations (Step 5). Finally, we collect the migration-related code changes from the identified migrations (Step 6).

A. Step 1: Clone repositories

We use SEART [?] to fetch an initial list of Python GitHub repositories. We set the language to Python, exclude forks

[1]https://doi.org/10.5281/zenodo.7574849

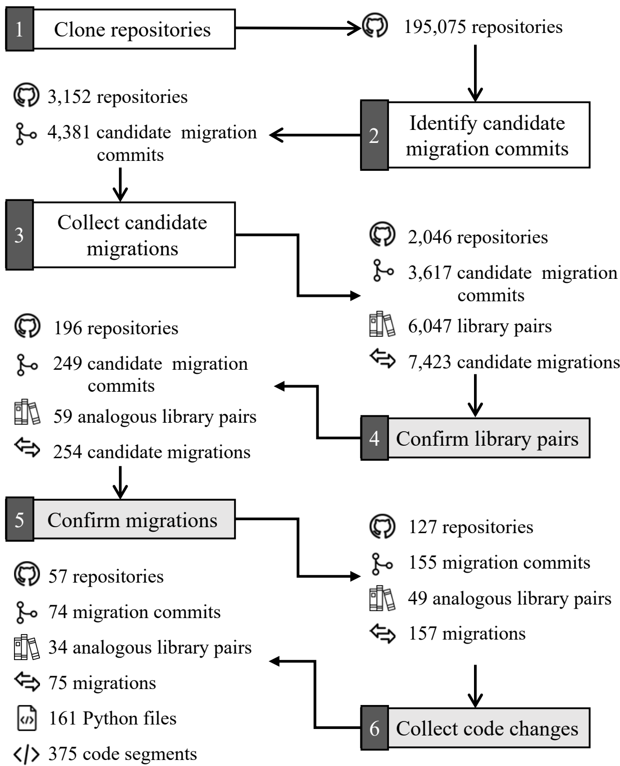

Fig. 1: The process of constructing PYMIGBENCH

to avoid redundant data, and retrieve 198,883 repositories. Note that SEART only includes repositories having 10 or more stars. Among the 198,883 repositories, we further discard 3,794 repositories that are no longer publicly available and 14 repositories that we could not clone due to miscellaneous errors. We clone the remaining 195,075 repositories.

B. Step 2: Identify candidate migration commits

We automatically analyze the commits of the cloned repositories to find the candidate migration commits. We use the following heuristics to identify such commits while excluding as many non-migration commits as possible. (1) *Not a merge commit:* we exclude merge commits because changes in merge commits are already reflected in the parent branches that we analyze. (2) *Dependencies have changed:* since migration is a replacement of one library with another, the commit must have both additions and deletions of lines in the *dependency file* (a file where the library dependencies are stored). We consider *requirements.txt*, *environment.yml*, *pipfile* and *pyproject.toml* as dependency files. (3) *Not a bot-created commit:* we ignore such commits because bot-generated migration commits typically relate to version updates and do not change the code.

We mark a commit as a candidate migration commit if all the above criteria are true. We use PyDriller[2] to check the first two criteria. To identify bot-created commits, we collect commit authors whose names contain the term "bot" and manually check the authors' profiles to verify if they are

[2]https://pydriller.readthedocs.io

```
- coverage==4.5.4
+ coverage>=5.2.1
- nose>=1.3.7
+ pytest-cov==2.11.1
+ pytest==6.2.0
```

Listing 1: Sample changes in a requirements.txt file

bots. The above criteria result in 4,381 candidate migration commits from 3,152 client repositories.

C. Step 3: Collect candidate migrations

Now we automatically analyze changes in the dependency files of each of the 4,381 commits to identify potential library replacements, i.e., *candidate migrations*. Note that more than one pair of libraries can be replaced in one commit, resulting in multiple migrations in a single commit.

We first collect the sets of added and deleted libraries from the dependency file associated with a candidate migration commit. We then remove the libraries that are present in both sets, ignoring their versions, because we are looking for migrations between different libraries. Finally, the cross-product of these two sets is the set of candidate library migrations for this commit. For example, in the changeset in Listing 1, *(nose, pytest-cov)* and *(nose, pytest)* are the candidate library migrations.

We find that the 4,381 candidate commits have 23 candidate migrations per commit on average, which seems like a large number of migrations in a single commit because a median commit has only one candidate migration. After some manual inspection, we identify that some commits add and delete a lot of dependencies producing a large number of candidate migrations with no real migration happening in those commits. We use Tukey's fences outlier detection technique [**?**] and exclude commits that have more than 9 candidate migrations. After discarding outlier commits, we are left with 3,617 candidate commits having 7,423 candidate migrations between 6,047 unique library pairs. On average, this remaining data has 2 candidate migrations per commit.

D. Step 4: Confirm library pairs

For a migration to be valid, it must be between an *analogous library pair*, i.e., two libraries that provide similar functionality allowing one to be replaced with the other. Therefore, instead of manually verifying all 7,423 candidate migrations, we first identify which of the 6,047 library pairs are analogous. We notice that most of the library pairs (5,826) have only one or two candidate migrations. To increase our chances of reviewing meaningful library pairs, we focus on reviewing 221 pairs with 3 or more candidate migrations.

To confirm that a library pair is analogous, the first two authors, each having 6 years of industrial experience, independently review the library pairs and discuss any disagreements. We achieve 0.85 Cohen's kappa score, which means almost perfect agreement [**?**]. We look into various documentation sources (e.g., PyPI pages, official websites, and the libraries' GitHub repositories) to find if they explicitly mention that the

two libraries are alternatives or that the libraries implement similar functionality. During our manual review, we find that some libraries do not offer an API that can be used in client code such as *pyflakes*[3] and *sphinx-material*[4]. Since we are primarily interested in migrations requiring client code changes, we discard such libraries. Additionally, we discard testing libraries, because we are interested in application migrations. We also identify the domains of confirmed analogous library pairs. The first two authors again independently label the library domains. We do not use any predefined set of domain names; therefore the two authors come up with different names in many cases. The first three authors then discuss the labels to create a final list of domains (available in our artifact).

We find that 59 of the 221 library pairs are analogous and offer APIs. These 59 library pairs are from 13 domains and appear in 254 candidate migrations in 249 commits across 196 repositories. The 59 library pairs contain 99 unique libraries.

E. Step 5: Confirm migrations

We now manually examine the 249 commits containing the identified 254 candidate migrations to confirm the migrations. We consider a candidate migration in a commit as a valid migration if (1) the commit message or code comments in the commit explicitly indicate the migration, or (2) the commit has code changes clearly related to the migration. To verify the migrations, the first and second authors individually review them and then discuss the results to resolve any disagreements.

At the end of this process, we identify 157 valid migrations between 49 library pairs in 155 commits from 127 repositories.

F. Step 6: Collect migration-related code changes

A migration commit may contain changes that are not related to the migration. Therefore, to facilitate the use of PyMigBench for library migration research, we identify and record the migration-related code changes that developers make. For example, Figure 2 shows part of the commit diff for a migration from `requests` to `aiohttp`. The changes on the removed Lines 31-32 are not related to this migration, whereas changes on the removed Lines 35-36 are related since they modify code that originally used the source library. Therefore, we record removed lines 35-36 and added lines 36-40 as a migration-related code change *segment*.

The first and second authors individually go through each of the 155 confirmed commits to identify the migration-related code changes. The two authors then discuss the code changes to resolve any disagreements. If they cannot come to a conclusion, the third author joins the discussion and the three of them discuss the code changes. We find 75 of the migrations have migration-related code changes in 375 code segments across 161 Python files in 57 client repositories. These occur between 34 library pairs from 11 domains.

[3]https://github.com/PyCQA/pyflakes

[4]https://github.com/bashtage/sphinx-material

```
31      -     if re.search(regex, url):
32      -         result = regex.search(url)
    32  +     if re.search(url_regex, url):
    33  +         result = url_regex.search(url)
33  34            url = result.group(0)
34  35
35      -     page = requests.get(url)
36      -     soup = BeautifulSoup(page.content, 'html.parser')
    36  +     async with aiohttp.ClientSession(headers={'User-
            Agent': 'python-requests/2.20.0'}) as session:
    37  +         async with session.get(url) as response:
    38  +             page = await response.text()
    39  +
    40  +     soup = BeautifulSoup(page, 'html.parser')
```

Fig. 2: Part of the diff file for the migration in Listing 2.

Listing 2: A YAML file describing a migration

```
source: requests
target: aiohttp
repo: raptor123471/dingolingo
commit: 1d8923a
commit_message: Replace requests with aiohttp
code_changes:
- filepath: musicbot/linkutils.py
  lines: [1:1, 35-36:36-40, 98-99:100-104]
```

III. PyMigBench

PyMigBench includes data of analogous library pairs and library migrations. We store the data in YAML, diff, and Python files. All our data are text files and are arranged in a specific folder structure for easy manual navigation and access to the data. To facilitate automated use of the dataset, PyMigBench also comes with a command-line tool.

We store each analogous library pair in its own YAML file, which includes the names of the source and target libraries and the domain of the library pair.

We also provide a YAML file for each migration, which contains the names of the library pair, client repository, commit hash, modified files, and line numbers of modified code segments. Listing 2 shows the content of a YAML file for a migration from *requests* to *aiohttp* where the commit message explicitly mentions that a migration happened in this commit. It also shows that the code of the *linkutils.py* file was changed during the migration. Moreover, the code at lines 1, 35 to 36, and 98 to 99 in the file were replaced with code at lines 1, 36 to 40, and 100 to 104, respectively, during the migration.

For each Python file containing migration-related code changes, we provide a git diff file[5] that can be viewed with a standard diff viewer or any text editor. We also provide the version of the code file that uses the source library (i.e., code before migration) and the version that uses the target library (i.e., code after migration). Figure 2 shows part of the Python file `linkutils.py` that was modified during the migration from *requests* to *aiohttp*, described in Listing 2.

We provide a Python-based command line tool to allow automated use of the benchmark. A user can use the tool to

[5]https://git-scm.com/docs/git-diff#_combined_diff_format

view summaries of the benchmark (i.e. descriptive statistics of the included data) as well as to query the benchmark by different dimensions. For example, a user can query: *find all migrations in repo "voice2json"* or *find all migrations to target library "aiohttp"* or *find all migrations in commit 1d8923a*. This allows other researchers to slice the data for an evaluation or additional analysis according to the capabilities of the migration techniques they are designing. The PYMIGBENCH repository contains documentation about how to use the dataset and the accompanying toolchain.

IV. USES OF PYMIGBENCH

Facilitating library mapping research: We provide a collection of 59 manually validated analogous Python library pairs from 13 different domains. There are currently no techniques that automatically extract analogous Python library pairs. Researchers can use the library pairs available in PYMIGBENCH for evaluation or as training data for library mapping techniques. While the number of library pairs may be limited for certain types of techniques, researchers can use this data to identify interesting/differentiating features that can help build automated identification techniques.

Facilitating migration-related code transformation research: PYMIGBENCH has 75 migrations with code changes between 34 library pairs in 57 client projects. Library migration techniques can use these projects in their evaluation. Since we have already validated migration-related code changes in these projects, researchers can verify the effectiveness of library migration techniques by matching the transformed code with the corresponding code changes in PYMIGBENCH.

Overall, PYMIGBENCH helps researchers in developing library migration techniques and, equally importantly, in systematic and fair evaluation and comparison.

V. THREATS TO VALIDITY

Internal validity. We may have missed some potential migration data for the following reasons. (1) We excluded forks to avoid analyzing duplicate commits, but a fork may contain commits not present in the original repository. (2) We identified dependency files based on commonly used names, but developers are free to use any name for their dependency files. We also did not consider the dependencies that may be declared in setup.py. (3) Our process of identifying migrations assumes that the addition and deletion of dependencies happened in the same commit, which may not always be the case [?]. (4) When identifying code changes, we only look at the commit where the library dependency was updated. (5) Developers may not always remove a source library from the dependency file, even if they no longer use it. However, code changes may be done in later commits as well, therefore, we may have missed some migration-related code changes. Being able to identify all the above missing data may potentially increase the data in PYMIGBENCH. However, this does not affect the validity of our results, because our main goal is to build a benchmark of real migrations, not necessarily find all migrations that happened in repositories hosted on GitHub.

External validity. PYMIGBENCH has migrations between only 34 library pairs. However, our migration examples are from 11 different domains, which provides some diversity.

Construct validity. We manually review migrations to identify migration-related code changes, which relies on the authors' knowledge of the libraries. To minimize mislabeling, 2 authors each having 6 years of industrial experience reviewed each library's documentation to get sufficient knowledge about the library and then independently reviewed the data.

Even though there were candidate commits from 2,046 repositories, we reviewed commits from a sample of only 708 repositories. To check for sample bias and representativeness, we ran a Mann-Whitney U test and a Kolmogorov-Smirnov test to compare various characteristics (repo size, number of commits, branches, contributors, watchers, and stars) of the repositories in the full population and in our sample. We found no statistically significant differences except for number of commits and branches.

VI. RELATED WORK

Library migration benchmarks: Teyton et al. [?] developed a frequency based semi-automatic approach to detect analogous Java libraries. They applied their approach to libraries available in the Maven central repository and manually verified a set of 80 analogous library pairs. In a follow up work [?], they extend their dataset to 329 analogous Java library pairs from 32 different domains. He et al. [?] verified 1,434 analogous library pairs from candidate pairs generated by their analogous library recommendation system. In a followup study, they built a benchmark of 3,163 manually verified Java migration commits [?]. These are all Java benchmarks and not directly useful for Python library migration.

API mapping and client code transformation: Balaban et al. [?] developed a technique to migrate the uses of legacy Java classes. Teyton et al. [?] and Alrubaye et al. [?], [?] analyze existing migrations to identify API mappings. PYMIGBENCH can be used to develop similar techniques in Python. Chen et al. [?] use code and documentation to train a model for API mapping recommendation without existing migrations. Along the same lines, SOAR [?] identifies API mappings based on the textual similarity between API descriptions. SOAR also supports client code transformation using program synthesis. To the best of our knowledge, SOAR is the only technique that works for Python, although evaluated only on one Python library pair. PYMIGBENCH provides an opportunity for additional evaluation.

Version migration techniques: There are several research efforts to automatically transform client code to use a different version of a given library, both for Java [?], [?], [?], [?], [?], [?], [?] and Python [?], [?].

VII. CONCLUSION

In this paper, we presented PYMIGBENCH, the first Python library migration benchmark. PYMIGBENCH contains a manually verified dataset of 75 migrations having migration-related

code changes between 34 library pairs, and 157 migrations between 49 library pairs in total. PYMIGBENCH facilitates future research on Python library migration by enabling systematic evaluations and comparisons of library migration techniques. Researchers can use the data in PYMIGBENCH to derive insights for building new techniques and understanding the limitations of the existing techniques.

REFERENCES

[1] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 233–244.

[2] A. A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to api deprecation," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3824–3870, 2019.

[3] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 72–83.

[4] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2187–2200.

[5] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[6] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the apache software foundation projects," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 154–164.

[7] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. 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*, 2020, pp. 245–256.

[8] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 289–298.

[9] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014.

[10] C. Chen, Z. Xing, and Y. Liu, "What's spain's paris? mining analogical libraries from q&a discussions," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1155–1194, 2019.

[11] F. L. De La Mora and S. Nadi, "Which library should i use?: A metric-based comparison of software libraries," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 2018, pp. 37–40.

[12] R. El-Hajj and S. Nadi, "Libcomp: An intellij plugin for comparing java libraries," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1591–1595.

[13] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 192–201.

[14] H. Alrubaye and M. W. Mkaouer, "Automating the detection of third-party java library migration at the function level." in *CASCON*, 2018, pp. 60–71.

[15] 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 *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 347–357.

[16] 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 Transactions on Software Engineering*, vol. 47, no. 3, pp. 432–447, 2019.

[17] A. Ni, D. Ramos, A. Z. Yang, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues, "Soar: a synthesis approach for data science api refactoring," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 112–124.

[18] Z. Zhang, M. Pan, T. Zhang, X. Zhou, and X. Li, "Deep-diving into documentation to develop improved java-to-swift api mapping," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 106–116.

[19] H. Alrubaye, M. W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, and J. Mcgoff, "Learning to recommend third-party library migration opportunities at the api level," *Applied Soft Computing*, vol. 90, p. 106140, 2020.

[20] M. Lamothe, W. Shang, and T.-H. P. Chen, "A3: Assisting android api migrations using code examples," *IEEE Transactions on Software Engineering*, 2020.

[21] M. Fazzini, Q. Xin, and A. Orso, "Apimigrator: an api-usage migration tool for android apps," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, 2020, pp. 77–80.

[22] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of api migration edits," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 335–346.

[23] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering repetitive code changes in python ml systems," in *International Conference on Software Engineering (ICSE'22). To appear*, 2022.

[24] M. Dilhara, A. Ketkar, and D. Dig, "Understanding software-2.0: A study of machine learning library usage and evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–42, 2021.

[25] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for msr studies," in *2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 560–564. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00074

[26] J. W. Tukey *et al.*, *Exploratory data analysis*. Reading, Mass., 1977, vol. 2.

[27] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[28] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on java library migrations: prevalence, trends, and rationales," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 478–490.

[29] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 265–279, 2005.

[30] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 274–283.

[31] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[32] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 325–334.

[33] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 599–602.

[34] C. Zhu, R. K. Saha, M. R. Prasad, and S. Khurshid, "Restoring the executability of jupyter notebooks by automatic upgrade of deprecated apis," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 240–252.

[35] S. Alamir, P. Babkin, N. Navarro, and S. Shah, "Ai for automated code updates," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 25–26.