Empirical Software Engineering Journal (EMSE) manuscript No.
Preprint of EMSE-D-25-00904 – Accepted to appear

MigrateLib: A Tool for End-to-End Python Library Migration

Mohayeminul Islam · Ajay Kumar Jha · May Mahmoud · Sarah Nadi

Received: date / Accepted: date

Abstract Library migration is the process of replacing a library with a similar one in a software project. Manual library migration is time-consuming and error-prone, as it requires developers to understand the Application Programming Interfaces (API) of both libraries, map equivalent APIs, and perform the necessary code transformations. This paper presents an LLM-based end-to-end solution that can automatically migrate code between any arbitrary pair of Python libraries that provide similar functionality. Before building the technique, we first systematically study the capabilities of LLMs for library migration on PyMigBench, an existing benchmark of 314 real-world library migrations. We find that the best performing LLM, GPT-4o, can correctly perform 57% of the migrations. We also identify post-processing steps that can further improve the performance. Based on this, we develop MigrateLib, a command-line application that combines the power of LLMs, static analysis, and dynamic analysis to provide accurate library migration. We evaluate MigrateLib on 717 migration scenarios in 175 real-world Python applications that are not from PyMigBench. We find that MigrateLib can migrate 32% of the migrations with complete correctness without any developer intervention. We inspect a sample of the incomplete migrations to understand how many of the required changes have already been done by MigrateLib and how many manual changes are still needed to complete the migration. We find that for more than half of the inspected migrations, MigrateLib completed 73% of the required

M. Islam
University of Alberta
E-mail: mohayemin@ualberta.ca

A. Jha
North Dakota State University
E-mail: ajay.jha.1@ndsu.edu

M. Mahmoud
New York University Abu Dhabi
E-mail: m.mahmoud@nyu.edu

S. Nadi
New York University Abu Dhabi
E-mail: sarah.nadi@nyu.edu

migration-related changes, leaving only 27% of the required changes for manual intervention.

Keywords Library migration · Python · Large Language Models

1 Introduction

Library migration is the process of replacing a library with an alternative one in a software project. Once a developer decides on a new target library, library migration involves identifying the old library usage in the codebase, finding the corresponding Application Programming Interfaces (APIs) in the new library, and replacing the old API usage with the new API usage. This can be tedious and error-prone, especially when the library is extensively used in a large codebase (Kula et al., 2018).

Researchers have proposed various techniques to automate the library migration process. Most of these techniques, however, are limited to only finding the analogous APIs (*API mapping*) (Teyton et al., 2013; Alrubaye and Mkaouer, 2018; Alrubaye et al., 2019b; Chen et al., 2019; Zhang et al., 2020; Di Rocco et al., 2025), but do not replace the old API usage in the codebase with the new one (*code transformation*). To find analogous APIs, some of these techniques mine existing migrations (Teyton et al., 2013; Alrubaye and Mkaouer, 2018; Alrubaye et al., 2019b) while others use machine learning models (Chen et al., 2019; Di Rocco et al., 2025) and natural language processing techniques (Ni et al., 2021). To the best of our knowledge, there are two techniques that actually perform code transformation (Ni et al., 2021; Nikolov et al., 2025), but both techniques evaluate their approaches on a small set of selected library pairs without providing a tool that can be used in practice.

In recent years, Large Language Models (LLMs) have shown that they can be effective in various software engineering tasks (Ozkaya, 2023; Fan et al., 2023; Wang et al., 2024), including the necessary upstream tasks for library migration, such as code generation (Nguyen and Nadi, 2022; Peng et al., 2023; Murali et al., 2023), code comprehension (Nam et al., 2024), and code transformation (Wei et al., 2023; Xia et al., 2023; Zhou et al., 2023a). This suggests that LLMs may be capable of performing library migration. However, studies have also identified limitations of LLMs in other software engineering tasks, especially generating incorrect or incomplete code (Fan et al., 2023). These previous efforts suggest that LLMs may be capable of performing library migration and that LLM-based migration tooling could be effective. However, we still need further empirical evidence to support this claim.

In this paper, our goal is to create an end-to-end solution for Python library migration that can potentially leverage the power of LLMs. We do not consider the problem of deciding which library to migrate to, but rather focus on the actual migration process once the developer has already decided on the target library. We choose Python as the programming language for our study, as it is one of the most popular programming languages and has a rich ecosystem of libraries. We approach our goal in three phases as illustrated in Figure 1. In Phase 1, we first systematically investigate the capabilities and challenges of using LLMs for library migration. We conduct an experiment using 314 migrations from PyMigBench (Islam

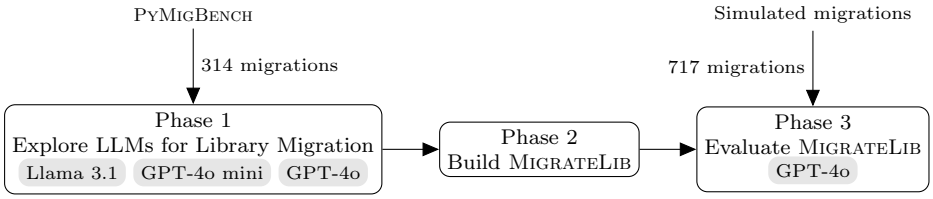


Fig. 1: Overview of the contributions.

et al., 2023), a Python library migration benchmark we previously built. We experiment with three LLMs (Llama 3.1, GPT-4o mini, and GPT-4o) and measure the correctness of the migrations by comparing the LLM's migrations with the original developer's migrations. We find that the three LLMs can correctly migrate 26%, 49%, and 57% of the migrations, respectively. Using an alternative test-based evaluation on a subset of the projects that have unit tests, we find that 36%, 52%, and 64% of the LLM migrations maintain the same test results as the original developer migrations for the three LLMs, respectively. We also study the types of migration-related code changes LLMs excel at vs. struggle with. Overall, we find that LLMs are capable of library migration, and as expected, the more powerful LLM (GPT-4o) performs best. We also identify four major challenges that make solely relying on LLMs in a practical tool less effective: (1) The files that need migration are not always known beforehand. (2) All LLMs often return only part of the code, skipping parts of the code that do not require migration. (3) For a given project, a file that we migrate may have other files that depend on it which may require changes during migration. Specifically, if any functions are made asynchronous during the migration, all parts of the code that use those functions must also be made asynchronous. (4) The inconsistencies between the versions between the libraries used in the project and those assumed by the LLM can lead to incorrect migrations.

In Phase 2, we build an LLM-based CLI tool, MigrateLib, that automates the full library migration process while addressing the previously identified limitations through pre- and post- processing using static and dynamic program analysis. MigrateLib takes in a complete Python project with an existing test suite and automatically performs the migration for the specified library pair. MigrateLib first identifies all files that require migration using static analysis and a dynamic call graph and sends those files to the LLM via a prompt that instructs the LLM to perform the migration. MigrateLib then verifies the results of the migration by comparing the test results before and after migration. If any previously passing tests now fail, MigrateLib performs a post-processing step where it checks if the LLM skipped any code and merges it back. It also detects whether any functions were made async during the migration, and if so, finds other parts of the project that use those functions and makes them async as well. Finally, it runs the unit tests again to determine the final migration status.

In Phase 3, we evaluate MigrateLib using a new purely test-based evaluation setup, as opposed to the benchmark-based evaluation we did in Phase 1. This allows us to conduct a larger-scale evaluation and also evaluate new migrations without the need for ground truth developer changes. We run MigrateLib on a set of 717 simulated migrations in 175 unique projects. We define a *simulated mi-*

gration as one that is not retrieved from the project's history. Specifically, we find source libraries used in a project and migrate the project to an analogous target library. We ensure that the projects we choose have high code coverage (above 80%), so that the evaluation is reliable. Given that GPT-4o had the best performance in Phase 1, we conduct this evaluation only using GPT-4o. We find that MigrateLib is able to successfully complete 32% migrations without any manual intervention (i.e., all tests that pass before migration still pass after migration). To understand how much manual work is required to fix the remaining 484 migrations that the LLM left in an incomplete stage, we randomly select a subset of 43 diverse migrations and attempt to manually fix them. We measure the amount of manual work required to complete these migrations using the number of lines of code we need to manually edit. We find that we are able to fix 24 (56%) of our sampled migrations, where the manual code modification is only 27% of the total changes needed for the migration.

Overall, our work provides a combination of empirical and engineering contributions that systematically address the challenges of library migration. This paper is an extended version of our prior conference paper (Islam et al., 2025) that only focused on Phase 1. Phases 2 and 3 are novel contributions of this work. To summarize, the contributions of this paper are:

1. An empirical study to explore the use of LLMs for library migration by comparing LLM changes to 314 benchmark developer changes. Our analysis systematically highlights the types of migration-related code changes LLMs excel at vs. struggle with.
2. We design and build an open-source end-to-end CLI tool, MigrateLib, that automates the full library migration process using a combination of LLMs and pre- and post-processing using static and dynamic program analysis.
3. We design a new evaluation setup that allows us to evaluate MigrateLib on a large scale and also on newer projects. We run an extensive test-based evaluation of MigrateLib on a set of 717 simulated migrations, and show that it works well with no or minimal manual editing for a diverse set of library pairs.

The source code of MigrateLib, the experimental data, and the detailed results of our experiments are available at `https://doi.org/10.6084/m9.figshare.29602517`.

2 Background and Terminology

In this section, we provide background information and terminology used in the rest of the paper.

2.1 Library migration

Library migration is the process of updating a software project to replace a used library with another one that provides similar functionality (Teyton et al., 2012). The *source* library is the one being replaced, and the *target* library is the one that replaces it (Teyton et al., 2012). One *migration* instance refers to a commit in a repository where a migration happened from a specific source library to a

target library (Islam et al., 2023), denoted using the notation *source→target*. For example, the commit `b0607a26` in repository *openstack/ironic* is a *retrying→tenacity* migration.

A *migration-related code change*, or *code change* for brevity, is a minimal replacement of source library APIs with target library APIs that cannot be meaningfully reduced further without losing the semantics of the change (Islam et al., 2023). A migration contains one or more code changes. The lines between the boxes in Figure 2 show code changes. For example, segment P1 in Figure 2a is replaced by segment D1 in Figure 2b.

We use subscripts *pre*, *dev*, and *llm* to denote data before migration, after developer's migration, and after an LLM's migration, respectively. For example, Code_{pre}, Code_{dev}, and Code_{llm} denote three states of a piece of code. Change_{dev} and Change_{llm} denote code changes by developers and an LLM, respectively.

2.2 PyMigBench and PyMigTax

We use two of our prior artifacts: PyMigBench (Islam et al., 2023) and PyMigTax (Islam et al., 2024), in this work which we briefly describe next. PyMigBench is a dataset of real-world Python library migrations, which we mined from open source GitHub repositories. We use the latest version of PyMigBench, version 2.2.5, in our experiments in this paper. The dataset has 321 migrations and 2,989 migration-related code changes. It includes a detailed description of each code change, including line numbers and API names, which facilitates our evaluation.

PyMigTax is a taxonomy of migration-related code changes (Islam et al., 2024), built using the first version of PyMigBench. In our prior work, we validated its generalizability on additional third-party data. PyMigTax describes a code change based on three dimensions: *(1) Program elements*: the types of program elements involved in the change (e.g., function call and attribute); *(2) Cardinality*: how many source APIs are replaced with how many target APIs. For example, one-to-many cardinality means one source API is replaced with multiple target APIs; one-to-many, many-to-one, and many-to-many cardinalities are commonly referred to as *higher-cardinality*; and *(3) Properties*: Additional properties to describe the code change (e.g., element name change when the source and target APIs have different names.). The code change data in PyMigBench is already annotated with the PyMigTax categories. We use PyMigTax categories (shown in Table 3) to understand the types of code changes LLMs can handle. The PyMigTax paper (Islam et al., 2024) has full category descriptions.

3 Phase 1: Exploring LLMs for library migration

We first systematically explore the capabilities of LLMs for library migration, while understanding how they handle different types of migration-related code changes. We first explain our setup and methods, and then present the results.

3.1 Experiment setup

We answer the following research questions in this phase:

Listing 1: LLM Migration Prompt

```
The following Python code uses library <source-lib>. Migrate this code to use
library <target-lib> instead. In the output, first explain the changes you made.
Then, provide the modified code.

Do not make any changes to the code that are not related to migrating between
these two libraries. Do not refactor. Do not reformat. Do not optimize. Do not
change coding style.

Provided code:
<premig-code>
```

RQ-1.1 **How similar are the LLM migrations to the benchmark migrations?** We consider the code changes stored in PyMigBench as the ground truth. We automatically check if the LLM was able to correctly perform all expected changes, while accounting for refactoring and alternative correct changes through manual review.

RQ-1.2 **How many migrations pass unit tests?** To evaluate run-time correctness of the migrated code, we use a second evaluation strategy where we run any available unit tests.

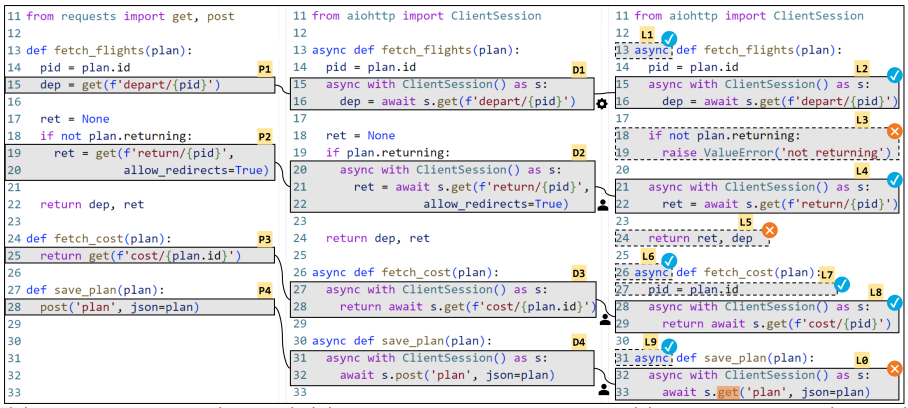
We start with the 321 migrations in PyMigBench for our experiments. We discard 2 migrations whose repositories are no longer public, and 5 migrations where either the pre-migration or post-migration files recorded in the benchmark have syntax errors. We conduct our experiments with the remaining 314 migrations containing 2,910 migration-related code changes.

3.1.1 Performing migration

We use three off-the-shelf LLMs – Llama 3.1-70B-Instruct (Dubey et al., 2024), GPT-4o-mini-2024-07-18, and GPT-4o-2024-08-06 (OpenAI, 2024) – to perform migrations on the PyMigBench dataset. We refer to these models as Llama, Mini, and 4o in the rest of the paper, respectively.

For each file having code changes in a given migration, we use the LLM prompt template in Listing 1 to migrate the code. The prompt is designed to ensure that the LLM performs focused, controlled migrations while maintaining transparency. By asking the LLM to explain its changes, we ensure that the LLM justifies its modifications, which can be useful for a human evaluator reviewing the migration. The prompt also aims to restrict the LLM from making unrelated modifications such that we remain focused on the task of library migration.

Since LLMs are non-deterministic (Atil et al., 2024), we use a temperature of 0 and run each migration 10 times for each model. To understand the extent of variability in the generated code, we calculate the difference between the runs. We use git-diff to compute a diff between each pair of the 10 runs and normalize the number of different lines by dividing it by the total number of lines in File_{pre}. We find that on average, two runs on the same file are only 6.6%, 4.0% and 4.8% different for Llama, Mini, and 4o, respectively. Given this low variability and the high effort involved in manual validation, we randomly select one run for each



(a) Before migration (File$_{pre}$) (b) Developer's migration (File$_{dev}$) (c) LLM Migration (File$_{llm}$)

Fig. 2: A sample *requests→aiohttp* migration.

The lines between the boxes show matching code changes. ⚙ denotes automatic matching, while 👤 denotes manual matching. Dashed lines denote changes that are not recorded in PyMigBench. The blue check marks denote correct changes, while the orange crosses denote incorrect changes.

model to evaluate the LLMs' results. Our artifact contains all LLM migrations for all 10 runs.

3.1.2 Matching LLM migrations with developer migration

In RQ-1.1, we compare the LLM migrations with the developer migrations in PyMigBench. We do this for each code change, and then aggregate the results at the migration level, while also considering the additional non-migration-related changes made by the developer and the LLM. We now explain the process using the example in Figure 2 that shows a migration from the library *requests* to *aiohttp*.

A. Auto change matching: We first attempt to automatically match as many LLM changes (Change$_{llm}$) as possible with developer change (Change$_{dev}$), while ensuring precision i.e., not falsely marking an incorrect Change$_{llm}$ as correct. Therefore, we consider only *exact* syntactic matches between Change$_{dev}$ and Change$_{llm}$, ignoring formatting differences. We find exact matches by identifying the AST nodes related to Change$_{dev}$ in File$_{dev}$ and then matching them to corresponding nodes in File$_{llm}$. If there are potentially multiple matches, we check the containing function and use proximity based heuristics demonstrated below. We ensure that the matched node sets translate to the same code string using `ast.unparse`.

Consider Change$_{dev}$ `D1` in Figure 2 which uses `ClientSession()` and `get()` from *aiohttp*. In File$_{llm}$, these APIs appear in multiple locations, but only Lines 15, 16, 21, and 22 are in the same containing function `fetch_flights` as Change$_{dev}$. Based on line number proximity, we identify two node sets `L2` and `L4` as potential matches for this Change$_{dev}$. By comparing the code strings, we find that `L2` exactly matches Change$_{dev}$. By the end of the auto change matching step, we get a set of matched pairs of Change$_{dev}$ and Change$_{llm}$, a set of unmatched Change$_{dev}$ (uChange$_{dev}$), and a set of unmatched Change$_{llm}$ (uChange$_{llm}$).

B. Manually reviewing unmatched changes: The automatic matching ensures that no change is incorrectly matched, but it may miss some correct matches due to syntactic differences, or alternative correct implementations. The manual review step focuses on reviewing uChange_{dev} and uChange_{llm}. We use the online API documentation as well as the source code of the source and target libraries to understand the correct API usage in the context of the file. For Figure 2, we manually find that `L4` is a correct alternative to `D2`, noting that the LLM omitted the default value (`True`) of the `allow_redirects` argument, unlike the developer. `L8` is a correct replacement of `D3`, where the LLM performed an extract variable refactoring (assigning `plan.id` to variable `pid` and then using `pid` in subsequent lines). However, we find that the LLM used the function `get` instead of `post` at line 33, making `L0` an incorrect alternative of `D4`.

After trying to match all changes in Change_{dev}, we may find that some Change_{llm} remain unmatched. Our aim is to categorize these as refactoring (not altering program semantics) or non-refactoring (potentially affecting code behavior). If a Change_{llm} is refactoring, we remove it from uChange_{llm}, as it does not impact migration correctness. In the example, the LLM introduced handling of a `ValueError` (`L3`), which changes the program behavior, so we keep it in uChange_{llm}. Similarly, swapping variables in `L5` is also non-refactoring.

Note that while PyMigBench records all migration-related code changes, it does not record additional changes *indirectly* related to the migration, i.e., lines that do not have a target library API usage. For example, adding `async` to function definitions (`L1`, `L6`, and `L9`) due to async calls introduced by migration are not recorded. We remove these changes from uChange_{llm} and mark them as correct. Overall, `L3` and `L5` stay in uChange_{llm}, while `L1`, `L6`, `L7`, and `L9` get removed.

We perform a multi-reviewer manual review to ensure the quality of the matching process. The first author first manually reviews 190 code changes from one model. Two other authors then review about half of those code changes each independently. We then measure the agreement using Cohen's Kappa (Cohen, 1960) to quantify the level of agreement between the reviewers. We do not reach substantial agreement, so we discuss and resolve disagreements and update our coding guideline. The two pairs of reviewers then independently review another 111 code changes, achieving a Cohen's Kappa score of 0.83 (almost perfect) and 0.73 (substantial) (Landis and Koch, 1977). Accordingly, we proceed with only the first author reviewing the remaining changes.

C. Determining code change status: Based on the matching of Change_{dev} and Change_{llm}, we determine each *code change status* as follows (example changes refer to Figure 2).

- *Correct change*: LLM's change is correct, either exactly like the developer (e.g., `D1`–`L2`), or with an alternative API (e.g., `D2`–`L4`), or with same APIs but with some refactoring (e.g., `D3`–`L8`).
- *Incorrect change*: The LLM incorrectly implemented the migration change: Used an incorrect API (e.g., `get()` instead of `post()` in `L0`), did not attempt to migrate it at all, or incorrectly removed part of a code.

D. Determining migration status: Based on the individual matched code changes, we now automatically determine an overall *migration status*:

- *Response failure*: The LLM did not generate a File_{llm} for at least one File_{pre} (e.g., due to token limit or API timeout).
- *Syntax error*: The LLM generated a File_{llm} for all File_{pre}, but at least one File_{llm} has syntax errors.
- *Incorrect*: The LLM could not correctly migrate *any* of the changes marked in PyMigBench. We assign this status when all Change_{dev} across the migration remain unmatched.
- *Partially correct*: The LLM correctly migrated only *some* of Change_{dev}. We assign this status when there are only some unmatched Change_{dev}.
- *Correct with non-refactoring changes*: The LLM correctly migrated *all* Change_{dev} but also performed some non-refactoring changes. We assign this status when there are no unmatched Change_{dev} in any file, but there are some remaining unmatched Change_{llm}.
- *Correct*: The LLM correctly migrated *all* Change_{dev} for this migration, without any non-refactoring changes. We assign this status when there are no unmatched Change_{dev} or Change_{llm} left in any of the files after the matching process.

3.1.3 Evaluating LLM migrations using unit tests

In RQ-1.2, we assess the same LLM PyMigBench migrations from RQ-1.1, but using the unit tests available in the corresponding repositories. A test-based evaluation that runs the code ensures that the migration preserves the original behavior. We now explain the process below.

We make a copy of the repository after the developers' migration (Code_{dev}). We then make another copy of developer's migration, but replace all File_{dev} with File_{llm}; we refer to this as Code_{llm}. The idea is that Code_{llm} is basically the version of Code_{dev} where the LLM did the migration on behalf of the developer.

To set up a virtual environment, we need to identify the Python version used in the client repository. If this information is available in the `setup.py` file, we use that version. Otherwise, we resolve the version based on the migration commit date as follows. We first identify the release dates of all minor versions of Python (3.6, 3.7 etc). Then, we find the latest release date that is before the migration commit date. This is the latest Python version that was available at the time of the migration; we use this version to create the virtual environment. Next, we install the code dependencies using `pyproject.toml`, `setup.py` and requirements files. In cases where specific dependency versions are not specified in those files, we look up the version history of the dependency on PyPI and install the latest version available at the migration commit date, similar to how we resolve the Python version. This ensures that the dependencies are compatible with the code at the time of migration.

Once the virtual environment is ready, we run all the unit tests while measuring coverage in the repository on Code_{dev}. If the run has errors, we read the error log and try to fix the errors, and run the tests again. Note that we only fix configuration or environment-related errors, not code errors. For example, if a dependency is missing, we install it; if the project requires a specific Python version, we set up the virtual environment with that version. We maintain a configuration file where we record any project-specific configurations or commands we used. We include this information in our artifact.

We find that 218 out of the 314 migrations have at least one test. Among these, 172 migrations had unresolvable errors. The failures are primarily due to missing

dependencies, especially for older projects. The remaining 46 migrations have tests that we are able to successfully run on Code_{dev}.

However, for the tests to be useful to validate migration, they must cover the migration-related code changes. Using the coverage report, we find that only 27 migrations have at least one test that covers the code changes, and among them, 25 migrations have at least one test that passes on Code_{dev}. Thus, the evaluation in RQ-1.2 focuses only on these 25 migrations. We run the entire test suite on Code_{llm} for these 25 migrations for each model using the same randomly selected LLM run used in RQ-1.1.

We compare the test results between Code_{dev} and Code_{llm} and assign the following statuses to the migrations:

- *Correct*: All passing tests in Code_{dev} also pass in Code_{llm}.
- *Partially correct*: Some of the tests that pass in Code_{dev} pass in Code_{llm}, but the others fail or raise run-time errors.
- *Incorrect*: None of the passing tests in Code_{dev} pass in Code_{llm}; they all fail or raise run-time errors.

3.2 RQ-1.1 How similar are the LLM migrations to the benchmark migrations?

We now discuss the results of RQ-1.1. We first present the migration level results, then present the overall code change level results, and finally analyze the correctness of code changes by PyMigTax categories.

3.2.1 Migration level correctness

Table 1 shows the migration level results. We find that 4o performs the best with 94% of the migrations having at least one correct code change, closely followed by Mini with 93%. When considering fully correct migrations, 4o outperforms Mini considerably, with 57% compared to 49%. Llama has the lowest performance, with only 26% being fully correct and 51% having at least one correct code change. This is mainly due to a higher proportion of *response failure*s where Llama ran into token limits. Llama also produces more syntax errors compared to the other two models (5.1% vs. 1.9% and 1.0%).

RQ-1.1: Migration level

All three LLMs were able to perform correct migrations with 4o performing best: 94% of its migrations were partially correct and 57% were fully correct.

3.2.2 Overall code change level correctness

The 314 migrations we use for our evaluation contain a total of 2,910 code changes. However, when there is a response failure or syntax error in the LLM's migration, we cannot evaluate the correctness of the corresponding code changes in that file. Accordingly, in this analysis level, we exclude 1,999 code changes from Llama, 944

Table 1: Correctness of PyMigBench migrations (RQ-1.1, migration level)

	Number (percentage) of migrations		
Status	Llama 3.1	GPT-4o mini	GPT-4o
At least partially correct			
Correct	83 (26%)	154 (49%)	179 (57%)
Correct w/ non-refactorings	28 (8.9%)	53 (17%)	55 (18%)
Partially correct	48 (15%)	84 (27%)	62 (20%)
Subtotal	*159 (51%)*	*291 (93%)*	*296 (94%)*
Fully incorrect			
Incorrect	7 (2.2%)	6 (1.9%)	2 (0.6%)
Syntax error	16 (5.1%)	6 (1.9%)	3 (1.0%)
Response failure	132 (42%)	11 (3.5%)	13 (4.1%)
Subtotal	*155 (49%)*	*23 (7.3%)*	*18 (5.7%)*
Total migrations	314 (100%)	314 (100%)	314 (100%)

Table 2: Correctness of PyMigBench migration-related code changes (RQ-1.1, overall code change level)

	Number (percentage) of code changes		
Status	Llama 3.1	GPT-4o mini	GPT-4o
Correct	808 (89%)	1,744 (89%)	1,867 (94%)
Incorrect	103 (11%)	222 (11%)	129 (6.5%)
Evaluated code changes	911 (100%)	1,966 (100%)	1,996 (100%)
Excluded code changes	1,999	944	914
Total code changes	2,910	2,910	2,910

from Mini, and 914 from 4o. This leaves us with 911, 1,966, and 1,996 code changes to evaluate for the three LLMs, respectively.

Table 2 shows the overall correctness of the LLMs' migrations at the code change level. 4o performs best, with 94% of the code changes being correct. Interestingly, Mini and Llama both perform equally well at the code change level, with both having 89% correct code changes. This is in contrast to the migration level, where Mini performs better than Llama (49% vs. 26%). This suggests that while Mini attempted more code changes compared to Llama, the ones that Llama was able to attempt without failures or syntax errors were comparatively correct.

RQ-1.1: code change level

LLMs correctly migrate a high proportion of most individual code changes, with 4o achieving 94% correct code changes. The other two models perform equally well with 89% correct code changes for both models.

3.2.3 Code change correctness by category

To understand if some types of code changes are more difficult to migrate, Table 3 shows the distribution of code change correctness across the different PyMigTax categories. The three columns in group "Frequency in PyMigBench" shows how often each PyMigTax category appears in all migrations (Migs), library pairs

Table 3: Correctness of code changes across PyMigTax categories (RQ-1.1, code change level by category)
Migs = Migrations, *LPs* = Library Pairs, *CCs* = Code Changes, *#Eval* = Number of code changes evaluated, *%Cor* = percentage of correct code changes.

PyMigTax category	Frequency in PyMigBench			Llama 3.1		GPT-4o mini		GPT-4o	
	Migs	LPs	CCs	#Eval	%Cor	#Eval	%Cor	#Eval	%Cor
Program elements									
Function call	229 (71%)	105 (78%)	1,804 (60%)	518	88%	1,093	87%	1,092	93%
Import	318 (99%)	132 (99%)	732 (24%)	276	95%	580	94%	593	98%
Decorator	39 (12%)	13 (10%)	411 (14%)	197	87%	335	90%	353	87%
Attribute	54 (17%)	33 (25%)	183 (6.1%)	63	83%	103	78%	103	89%
Type	17 (5.3%)	13 (10%)	55 (1.8%)	12	75%	34	85%	34	88%
Exception	16 (5.0%)	13 (10%)	29 (1.0%)	17	65%	29	86%	29	93%
Function ref	4 (1.2%)	4 (3.0%)	12 (0.4%)	3	33%	12	58%	12	100%
Properties									
NoProps	57 (18%)	36 (27%)	262 (8.8%)	43	84%	150	88%	150	97%
ElemNC	188 (59%)	88 (66%)	1,025 (34%)	398	84%	773	85%	771	90%
ArgAdd	93 (29%)	49 (37%)	578 (19%)	150	85%	244	80%	244	88%
ArgDel	66 (21%)	37 (28%)	349 (12%)	175	85%	234	87%	234	89%
ArgTrans	81 (25%)	38 (28%)	317 (11%)	115	77%	201	76%	220	77%
ParamAdd	13 (4.0%)	1 (0.7%)	127 (4.2%)	110	87%	127	95%	127	90%
ArgNC	17 (5.3%)	11 (8.2%)	112 (3.7%)	20	60%	58	59%	58	84%
AsyncTrans	13 (4.0%)	4 (3.0%)	50 (1.7%)	35	83%	50	90%	50	92%
OutTrans	16 (5.0%)	15 (11%)	49 (1.6%)	25	88%	49	78%	49	92%
Cardinality									
One-to-One	200 (62%)	98 (73%)	1,552 (52%)	408	86%	877	84%	894	91%
One-to-Zero	50 (16%)	25 (19%)	297 (10%)	99	99%	290	99%	290	100%
Zero-to-One	35 (11%)	18 (13%)	198 (6.6%)	34	97%	55	87%	55	89%
Higher Card.	*91 (28%)*	*45 (34%)*	*210 (7.0%)*	*94*	*70%*	*164*	*79%*	*164*	*84%*
One-to-Many	52 (16%)	23 (17%)	142 (4.8%)	61	69%	103	78%	103	80%
Many-to-One	30 (9.3%)	24 (18%)	41 (1.4%)	16	62%	34	82%	34	91%
Many-to-Many	21 (6.5%)	11 (8.2%)	27 (0.9%)	17	82%	27	78%	27	89%
Total	321	134	2,989	911	89%	1,966	89%	1,996	94%

(LPs) and code changes (CCs) in PyMigBench, which provides perspective on the category's impact in practice. This information is the same for each LLM since it describes the existing data in PyMigBench. For example, 1,804 (60%) of all 2,989 code changes involve function calls. The next three column groups show the performance of each model. Since each LLM has a different number of evaluated code changes, we show the number of evaluated code changes from each category (#Eval) alongside the proportion of these code changes that are correct (%Cor). For example, the *Function call* row under the *Program elements* group shows that Llama's syntactically correct migrations included 518 code changes that involve function calls, and that 88% of these code changes were correctly migrated.

A. Program elements: Function calls are the most common program elements in terms of code changes (60%), and second most common in terms of migrations (71%) and library pairs (78%). All three LLMs correctly migrate a high proportion of these changes, with 88%, 87%, and 93% of the function calls being correctly migrated by the three LLMs. Almost all migrations and library pairs in PyMigBench require migrating import statements, a task that all three LLMs perform well, especially 4o correctly migrates nearly all of them (98%).

Mini slightly outperforms the other models in migrating decorators (90% vs. 87% and 87%); however, it does worst in attributes (78% vs. 83% and 89%). 4o performs better than other models in migrating types, exceptions and function references, followed by Mini. Llama and Mini both struggle with migrating function references (33% and 58% correct), while 4o correctly performs all such code changes. While this last category shows most discrepancy between the models, this program element is present in only 0.4% of the code changes, therefore does not affect the overall performance significantly.

B. Properties: The property *no properties* indicates that the source and target APIs are identical, and hence no changes are required. While 4o correctly leaves almost all of these APIs unchanged (97%), Llama and Mini incorrectly changed several of them, resulting in 84% and 88% correct, respectively.

Element name change is the most common property in PyMigBench. This indicates that the target APIs commonly bear different names from the source APIs. All LLMs perform well in making this change, with Llama, Mini, and 4o correctly migrating 84%, 85%, and 90% changes, respectively.

All three LLMs performed better or equally well in *argument deletion* compared to those with *argument addition*, which is expected as adding an argument requires the LLM to find a suitable replacement, while deleting an argument is a simpler task. The *Argument transformation* property indicates that an argument requires various changes, such as changing the type or value. The three LLMs correctly migrate only 76-77% of the argument transformation changes.

The property *parameter addition to decorated function* is found in migrations in just one library, *click* library, where some decorators require adding a parameter to the decorated function (Pallets, 2024). All three LLMs perform well in migrating these changes; interestingly, Mini performs better than 4o (95% vs. 90%). Mini also performed better than 4o in migrating decorators, which are common in the click library. These are the only two categories where 4o does not perform the best.

Argument name change applies when an argument representing the same semantics has different names in the source and target APIs. While 4o performs reasonably well in migrating these changes (84%), Llama and Mini fail frequently with them (only 60% and 59% correct).

C. Cardinality: One-to-One code changes are the most common in PyMigBench, and all three LLMs perform well in migrating them. One-to-Zero code changes are the ones where a source API needs to be removed, but no target needs to be added. Because it is a simple delete operation, all three LLMs migrate all or almost all of these changes correctly. Its counterpart, Zero-to-One code changes, are the ones where there is no source library usage, but a target library usage needs to be added. This is a more difficult change, as the LLM needs to find a suitable location to add the new library usage. Interestingly, Llama performs the best in migrating these changes (97% correct), making this the only category where it performs better than the other models (87% and 89% correct). While previous research showed that higher-cardinality code changes are generally difficult compared to one-to-one code changes (Alrubaye and Mkaouer, 2018; Wang et al., 2016; Zhang et al., 2020; Huang et al., 2024), the PyMigBench dataset shows that higher cardinality

Table 4: Correctness of 25 PyMigBench migrations using their available unit tests (RQ-1.2)

	Number (percentage) of migrations		
Status	Llama 3.1	GPT-4o mini	GPT-4o
Correct	9 (36%)	13 (52%)	16 (64%)
Partially correct	1 (4.0%)	5 (20%)	2 (8.0%)
Incorrect	15 (60%)	7 (28%)	7 (28%)
Evaluated	25	25	25

changes are frequent with 34% of the library pairs requiring at least one higher-cardinality code change. Overall, we find that the LLMs are reasonably successful at migrating them, with 70%, 79%, and 84% correctness rates.

RQ-1.1: Code change level by category

LLMs correctly migrate a high proportion of most code changes, with 4o achieving 94% correct code changes. The models perform reasonably well in migrating difficult higher-cardinality code changes (4o gets 84% correct), but struggle relatively more with argument transformation and argument name change.

3.3 RQ-1.2 How many migrations pass unit tests?

Table 4 shows the results for the 25 migrations. 4o has the highest percentage of correct migrations, with 64%, followed by Mini with 52%, and Llama with 36%. Mini has more partially correct migrations than 4o and Llama, leading it to have an equal number of incorrect migrations as 4o (28%). Llama, on the other hand, has a much higher proportion of incorrect migrations (60%).

RQ-1.2

Out of 25 PyMigBench migrations with unit tests covering the migration-related code changes, the LLMs correctly migrated 36%-64%, with 4o being the highest.

3.4 Strengths of LLM for library migration

Overall, the results of this empirical study highlight the high overall correctness of LLM migrations. We now discuss the key strengths we observe.

Higher cardinality code changes: As shown in RQ-1.1, Llama, Mini, and 4o successfully migrated 70%, 79%, and 84% of the higher cardinality code changes, respectively, which the literature always viewed as complex migrations (Xu et al., 2019; Alrubaye et al., 2020). Figure 3 shows an example of a many-to-many migration from *twitter* to *tweepy* done by Llama. In addition to correctly replacing

Before migration (using *twitter*)

```
tw_api = Twitter(
  auth = OAuth(oauth_token, oauth_secret,
         consumer_key, consumer_secret))
```

Llama's migration (using *tweepy*)

```
auth = OAuthHandler(
    consumer_key, consumer_secret)
auth.set_access_token(
    oauth_token, oauth_secret)
tw_api = API(auth)
```

Fig. 3: Llama correctly migrating a many-to-many code change

Before migration (using *argparse*)

```
def parse_commands():
  p = ArgumentParser()
  p.add_argument("src", type=Path)
  p.add_argument("dst", type=Path)
  p.add_argument("usedir",
                 action = "store_true")
  return parser.parse_args()

def main():
  args = parse_commands()
  convert(args.src, args.dst, args.usedir)
```

4o's migration (using *click*)

```
@argument("src",
          type=click.Path())
@argument("dst",
          type=click.Path())
@argument("usedir", type=bool)
def main(src, dst, usedir):
  convert(src, dst, usedir)
```

Fig. 4: 4o correctly migrating between libraries that use different API styles.

Before migration (using *eventlet*)

```
socketio = SocketIO(app)
bootstrap = Bootstrap(app)
```

4o's migration (using *gevent*)

```
socketio = SocketIO(app, async_mode="gevent")
bootstrap = Bootstrap(app)
```

Fig. 5: 4o performing a correct inferred change

the functions `Twitter()`, and `OAuth()` with `OAuthHandler()`, `set_access_token()`, and `API()` with correct arguments, Llama also splits the code into multiple statements, making the code more readable.

Different API styles: We find that the LLMs are able to handle transformation between different API styles. For example, the library *argparse* provides API functions for parsing command line arguments, while *click* provides decorators. Figure 4 shows a migration example between these libraries. Despite the completely different API styles, the three LLMs correctly migrate 94% of code changes between this library pair.

Inferring changes outside of the source and target APIs: We find that LLMs can infer some additional changes beyond the immediate APIs of the source or target library. For example, the migration in Figure 5 adds the `async_mode` argument to the `SocketIO()` function. This is interesting because the API is from the *Flask-SocketIO* library, which is not the source (*eventlet*) or target (*gevent*) libraries. However, the `SocketIO` function works with several libraries, *eventlet* being the default (Flask, 2024). This means, if 4o did not explicitly set the `async_mode` argument to `gevent`, the code would break after the migration.

3.5 Challenges of using LLMs for library migration

From the results of the empirical study in Phase 1 and the above strengths, we see that LLMs have strong potential for library migration. However, we also noted several challenges that would need to be addressed if LLMs are to be used for practical tooling for library migration.

Challenge-1 Files requiring migration may not be known upfront: In our experiment, we use PyMigBench to identify the files that require migration that we will send to the LLM. However, in a real-world scenario, the files requiring migration may not be known upfront. Therefore, a tool needs to first identify the files that require migration before sending them to the LLM for migration.

Challenge-2 LLMs do not always return complete code: A common issue we observe across all three LLMs is that they often do not return the complete code. Instead, they skips parts of the code, usually commenting that the next block of code remains unchanged. This is especially when the skipped part does not require migration, despite our prompt explicitly asking them to return the entire code. While this is not necessarily incorrect, as the skipped code may not require any changes, the resulting code is incomplete and cannot be provided directly to the developer.

Challenge-3 Dependents of migrated files may also require changes: In our experiment, we provide the LLMs only the files that directly use the source library, as marked in PyMigBench. However, other files that depend on the migrated files may also require changes. This is especially true when functions are made asynchronous during the migration, as all parts of the code that use those functions must also be made asynchronous.

Challenge-4 Library version compatibility: We do not mention any library version requirements in the prompt (Listing 1), and we notice that the LLMs usually pick the APIs from the latest or recent versions of the target library. In few cases, this caused runtime errors when running tests in LLM's code, which we report as incorrect in RQ-1.2. This is because the application or the Python version used in the project may not be compatible with the latest version of the target library. In an actual migration scenario, the developers may want to migrate to a specific version of the target library that is compatible with their project.

4 Phase 2: Building MigrateLib

In Section 3, we show that LLMs can perform library migrations. However, we also identify several challenges when using LLMs for library migration, especially when building an *end-to-end* tool that developers can use. In this section, we describe Phase 2 of our work, where we build MigrateLib while addressing the identified challenges through pre- or post-processing steps as follows:

1. Before migration, MigrateLib identifies the files that use the source library, and thus require migration to address **Challenge-1**.
2. MigrateLib allows the user to specify the source and target library version, and include this in the prompt, to address **Challenge-4**.

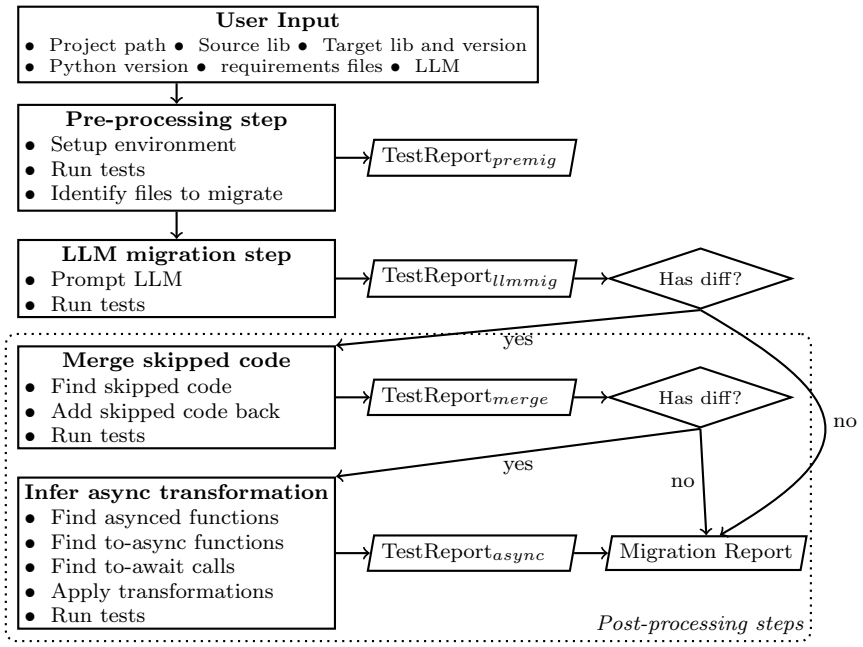


Fig. 6: Workflow of MigrateLib

The *Has diff?* nodes check whether a test report differs from TestReport$_{premig}$.

3. After migration, MigrateLib automatically adds back any skipped code to address **Challenge-2**.
4. After migration, MigrateLib identifies the functions that become asynchronous due to the migration, and finds all other functions in the projects that use these functions, and makes them asynchronous as well to address **Challenge-3**.

Figure 6 shows an overview of how MigrateLib works. MigrateLib works as a Command Line Interface (CLI) and takes the following as input: (1) the path to a Python project that needs library migration, (2) the source library name (the version is inferred), (3) the target library name and version, (4) the Python version in which the code should run, (5) paths to one or more requirements files, (6) and an LLM to use for the migration.

MigrateLib performs the migration in three main steps. First, it sets up a virtual environment for the project and identifies all files that require migration (*Pre-processing step*). In the next step, it prompts an LLM to migrate the identified files (*LLM migration step*). Finally, in the last step, it performs two post-processing steps to address **Challenge-2** and **Challenge-3** mentioned above to improve the migration results (*Post-processing step*).

We note that a key criteria for the input project is that it contains tests with high coverage, as MigrateLib uses the tests to verify the migration. While MigrateLib can perform the migration regardless of the test coverage, migration results without test validation may not be as reliable. MigrateLib also uses tests to complement the static analysis for identifying the files that require migration

(Task 3, Section 4.1 below), so the lack of tests may also impact the accuracy of the identified files.

4.1 Pre-processing step

The pre-processing step handles three main pre-processing tasks needed before the actual migration:

Task 1: Set up environment: MigrateLib creates a virtual environment for the provided Python version using the command `python -m venv .venv`, and then installs the required libraries using the `pip install -r <requirements-file-path>` command for each of the requirements files. It then installs the target library at the provided version using the `pip install` command. MigrateLib requires some additional packages, such as `pytest`, `pytest-cov`, which it also installs during this step. After the installation, MigrateLib derives the source library version by querying the installed package using the `pip show <source-library-name>` command.

Task 2: Run tests to get baseline: Once the environment is set up, MigrateLib runs the tests using the command `python -m pytest` on the project and saves the test report. We refer to this version of the test report as $TestReport_{premig}$, which is used as the baseline for the migration.

Task 3: Identify files that require migration: This task addresses **Challenge-1** from Section 3, where we need to identify the files that require migration before sending them to the LLM. One obvious way a file can use a library is by importing it. For example, in Figure 7, the file `main.py` imports the *geolib* library, thus it requires migration when migrating from *geolib* to another library. To identify such files, MigrateLib reads and parses all Python files in the project and looks for import statements of the source library. Note that a library can be imported using a different name than the library name; a library may also have multiple import names. Therefore, MigrateLib first finds all import names of the source library using the *Johnnydep* library (Jeantine-Glenn, 2025), which extracts the import names from the wheel file of the library.

A file may also use a library API without directly importing it. An example is shown in Figure 7. Notice that the `to_dict()` function in the `serialize.py` takes as input a shape object from the *geolib* library, even though it does not import the library. It uses the `Shape.name` attribute and `Shape.area()` method, which are part of the *geolib* library. To identify such files, MigrateLib builds a run-time call graph of the project by running the tests with profiling. It then analyzes the call graph to find all files that call any function from the source library.

Overall, in Figure 7, MigrateLib will identify the file `main.py` requiring migration using static analysis, and `serialize.py` requiring migration using dynamic analysis. We will mark both the files as requiring migration and send them to the LLM for migration in the next step. There may be cases where the same file is identified by both static and dynamic analysis, which is not a problem as we will only send the file once to the LLM for migration. Note that we search all Python files in the project, including test files, for both static and dynamic analysis.

serialize.py

```
def to_dict(shape):
  return {
    "name": shape.name,
    "area": shape.area()
  }

```

main.py

```
from serialize import to_dict
from geolib import Square
import pprint

sq = Square(5)
pprint(to_dict(sq))
```

Fig. 7: Example of using a library without importing

Listing 2: Prompt template used by MigrateLib

```
The following Python code currently uses the library <source-lib> version
<source-ver>. Migrate this code to use the library <target-lib> version
<target-ver> instead.

**Instructions:**
1. **Explain the Changes**: Begin the output with a brief explanation of the
   specific changes you made to migrate from "<source-lib>" to "<target-lib>".
2. **Provide the Modified Code**: After the explanation, present the modified
   code. Provide the entire code after migration even if only a part of it is
   changed.

**Important Guidelines**:
- Only make changes directly related to migrating between "<source-lib>" and
  "<target-lib>".
- Do not refactor, reformat, optimize, or alter the original coding style.
- The code given to you is part of a larger application. Do not change the names
  of classes, functions, or variables, because it can break the application.

Original code:
<premig-code>
```

4.2 LLM migration step

MigrateLib can be configured to use any LLM that adheres to OpenAI's Chat Completions API format (OpenAI, 2024), which has become a standard API for most models. For each file that requires migration, MigrateLib generates a prompt using the template in Listing 2 and sends the prompt to the LLM. The prompt is based on our original prompt used in Section 4 (Listing 1), but enhanced with more specific instructions. For example, we address **Challenge-4** by including the versions of the source and target libraries as an optional input that can be included in the prompt. Rather than automatically infer potentially incorrect versions ourselves, we treat the library versions as input parameters that the user can specify (and which we then include in the prompt), which also gives the user more control over the migration process.

The LLM response contains the migrated code, which MigrateLib extracts and saves. When the LLM finishes migrating all files, MigrateLib replaces the original files in the project with the migrated files. It then runs the tests and generates a test report, which we refer to as $TestReport_{llmmig}$.

4.3 Post-processing Step 1: Merge skipped code

We design this post-processing step to address **Challenge-2** from Section 3. The goal is to add back any unchanged code that LLM skipped returning in its response. For each file that was migrated in the LLM migration step, MigrateLib generates a diff between the original file and the migrated file using Python's `difflib` module. Then it attempts to detect which of the hunks were entirely removed from the original file. We make the following assumption to identify a hunk of skipped code:

1. The hunk should not use source library APIs such that it is not a migration-related change. Therefore, MigrateLib checks if any of the removed lines in the hunk uses any source library API. If yes, the hunk is not considered skipped.
2. The hunk must mostly include removal of lines. Our reasoning is that if the hunk has a similar number of additions and removals, it is likely that the hunk is replacing some code with other code, likely performing migration. If a hunk has more than 90% of removed lines compared to total added+removed lines, we consider it mostly removal.
3. The hunk should not be too small. This assumption avoids incorrectly considering small migration-related removals as skipped code. We consider a hunk as too small if it has less than 9 lines of code.

We determine the above criteria thresholds through experimentation on a set of migration examples. If a hunk is identified as skipped code by the above criteria, MigrateLib adds back the removed lines from the original file to the migrated file. After finishing the merging of skipped code, MigrateLib runs the tests again and generates a new test report, which we refer to as $TestReport_{merge}$.

4.4 Post-processing Step 2: Infer async transformation

This post-processing step addresses **Challenge-3** from Section 3 by identifying functions that become asynchronous due to the migration, and making their transitive callers asynchronous as well. Below we describe the steps in detail using the example in Figure 8.

Identify asynced functions: We call a function *asynced* if it was synchronous before migration, but is asynchronous after the LLM migration. For a given migrated file, MigrateLib creates an abstract syntax tree (AST) of the code before migration and another AST after migration. It then finds the function definitions that are not asynchronous in the AST before migration, but are asynchronous in the AST after migration. Since the location (line number) of the function definition may change due to the migration, MigrateLib matches the functions by their qualified name. In Figure 8, the function `fetch_data` is asynced.

Identify to-async functions: If a function `foo` calls a function `bar`, and `bar` calls a function `fizz`, then `foo` is said to transitively call `bar` and `fizz`. For a given asynced function, MigrateLib finds all its transitive callers using the call graph, and marks them as *to-async*, meaning that the async keyword must be added to their function definition. For example, in Figure 8, the function `fetch_data` in `service.py` is asynced

Fig. 8: Example of applying LLM migration and then infer async transformation

The first row presents the original code before migration. The second row displays the code after LLM migration. The third row shows the after applying infer async transformation. The left column contains the main code, while the right column contains its associated test. Changes introduced by each migration step are highlighted in red.

after the LLM migration. This function is called by the function `test_fetch_data` in `service_test.py`, therefore, MigrateLib marks `test_fetch_data` as to-async.

Identify to-await calls: MigrateLib again uses the call graph to find all call sites of the asynced and to-async functions, and marks them as *to-await*, meaning that an `await` keyword must be added to these calls. For example, in the call to `fetch_data` in the function `test_fetch_data` (line 5, after LLM migration) is marked as to-await, because it is a call to an asynced function.

Upon finding all asynced functions, to-async functions, and to-await calls, MigrateLib applies the necessary code transformations using the *LibCST* library

(contributors, 2025). Additionally, MigrateLib decorates any asynced and to-async test functions with the `pytest.mark.asyncio` decorator, and installs the *pytest-asyncio* package to the virtual environment, which is required to run asynchronous tests. After finishing the async transformations, MigrateLib runs the tests again and generates a new test report, which we refer to as $TestReport_{async}$.

5 Phase 3: Evaluating MigrateLib

In this phase, we want to run a large-scale evaluation of MigrateLib. While using PyMigBench in Phase 1 allowed us to compare the LLM migrations to a developer-performed ground truth migration for systematic understanding of LLM capabilities, it has several limitations: (1) not all projects have tests, which makes an automated test-based evaluation infeasible/limited and (2) all the migrations in PyMigBench happened prior to the known training cutoff date of the three models (Late 2023). Therefore, there is a possibility that the LLMs are simply reciting the migrated code they have seen before (for this particular codebase). Accordingly, in Phase 3, we come up with a different evaluation set up that allows us to evaluate MigrateLib on a larger scale and on migrations that the LLMs is unlikely to have seen before, while still being able to automatically determine the correctness of the migration without manual evaluation. We resort to the idea of *simulated migrations* where we prioritize finding Python projects with high-coverage test suites. We then find source libraries used in the projects and migrate the project to use an analogous target library.

Our evaluation aims to answer the following research questions:

RQ-2.1 **How many migrations can MigrateLib perform correctly?** We measure how well MigrateLib performs on the simulated migrations without any manual intervention.

RQ-2.2 **How much further code changes are required to complete the incomplete migrations?** While the best case scenario is that MigrateLib can migrate the project without any issues, a more practical expectation is that MigrateLib would do most of the task correctly, but there are some parts that require manual intervention. In this RQ, we measure the remaining manual work required based on the size of remaining code changes.

5.1 Data collection and selected model

We want to find repositories that use some source library, and then migrate them to use an analogous target library. To make the test-based evaluation reliable, we want to select repositories whose tests have good code coverage. We start by downloading a list of 95,472 Python GitHub repository metadata using SEART (Dabic et al., 2021) service. We use the filters to select Python repositories that have at least one commit after January 1, 2024 to discard repositories that are not actively maintained. We also filter out repositories with less than 100 lines of code to ensure that the repositories have some code to migrate. We then filter out repositories using the following criteria before cloning them for further analysis:

Listing 3: Prompt template for finding candidate analogous libraries

```
Identify alternative Python libraries that offer similar functionality as
<lib_name>, allowing for potential replacement in applications. The alternative
libraries should be available on PyPI.
Produce the result as a JSON list, having only the library names. The produced
library names should match the PyPI package names.
```

i. **Large repositories:** Extra-large repositories would take too long to clone and analyze, so we filter out 30,563 repositories larger than 10MB, leaving us with 64,909 repositories.
ii. **No requirements files:** We use requirements files to build the virtual environment to be able to run the tests. We filter out 39,916 repositories that do not have a `requirements.txt` file in the root directory, leaving us with 24,993 repositories.
iii. **No test files:** Python test file names usually contain the term *test*. We use GitHub code search API[1] to query files that contain the term *test*, and filter out a repository if it does not have any such files. We filter out 12,558 repositories that do not have any test files, leaving us with 12,435 repositories.

When a repository passes the above heuristics, we clone it and attempt to run the tests. In this process, we filter out the repositories with the following criteria:

iv. **Could not run tests:** We attempt to set up a virtual environment for the repository using the requirements file and run the tests following Section 3.1.3 (Evaluating LLM migrations using unit tests). Due to various repository-specific requirements, e.g., Python version, system requirements, and incomplete requirements file, we are often unable to run the tests. We filter out 10,104 repositories that we could not run the tests for, leaving us with 2,314 repositories.
v. **Low passing rate:** We filter out the repositories that have less than 90% of the tests passing, which filters out 957 repositories, leaving us with 1,357 repositories.
vi. **Low coverage:** We filter out repositories that have less than 80% line coverage by the tests, which filters out 557 repositories, leaving us with 800 repositories.
vii. **Tests require user interaction:** We identify 2 repositories that require manual intervention and exclude them, leaving us with 798 repositories, which we use for evaluation. We store the commit SHA for the version of the repository we use to ensure reproducibility.

We then collect the dependencies of the 798 selected repositories by parsing their requirements files. This gives us an initial list of 1,421 unique source libraries. We exclude 12 libraries that are not found on libraries.io (Katz, 2020), leaving us with 1,409 libraries. To ensure that the libraries are commonly known, we filter out 602 libraries that have less than 100 dependents, and filter out 2 libraries that do not have a description, leaving us with 805 potential source libraries.

[1] GitHub code search API: `https://docs.github.com/en/rest/search/search#search-code`

For each of the 805 source libraries, we find a list of candidate analogous libraries using gpt-4o-mini-2024-07-18 with the prompt template in Listing 3. This gives us a total of 4,856 candidate library pairs. All of these pairs may not actually be analogous, i.e., they may not be suitable for migration, so we need to verify them. This manual task can be very time-consuming; therefore, we first automatically filter out some of the pairs. First, we filter out 4,215 pairs where the source library is used in less than 10 of the selected repositories. This helps us to keep the manual effort of evaluating analogous pairs manageable by only selecting pairs that appear in multiple projects in our data set. Then, we filter out 192 pairs with an unmaintained potential target library. Specifically, we exclude the target libraries that have not been updated since January 1, 2024. This leaves us with 449 pairs to manually verify.

During manual validation, we look into the library documentation and confirm 60 library pairs that are indeed analogous and can be potentially migrated. We exclude the remaining because they fall into one of the following categories (i) **Not analogous:** Some libraries are not analogous, e.g., they provide different functionality or have different APIs. (ii) **Test related libraries:** For example, *pytest*, *mock*, etc. We exclude these libraries because our evaluation process relies on the tests. Therefore, migrating the test libraries themselves may break our automated evaluation. (iii) **Non-API tools:** Some dependencies are command line interfaces. Since they are not used in the code, we ignore them. For example, *black*, *flake8*, etc. (iv) **Documentation tools:** Some dependencies are used for documentation generation. Since they are not used in the code, we ignore them. For example, *sphinx*, *mkdocs*, etc.

Now that we have a list of 798 repositories and 60 analogous library pairs, we want to find which repositories actually use the source libraries in our list. It could be the case that some repositories list a requirement that is not actually used in the code. Accordingly, we parse the source code of each repository and look for import statements that use the source libraries. We find that 175 repositories use at least one of the source libraries. Note that each source library can have one or more analogous target libraries (e.g., *requests→aiohttp* and *requests→httpx*). We use the term *unique migration instance* to refer to the combination of a repository, a commit, a source library, and a target library. We use the latest commit of each repository at the time of our experiment (January 2025) for this purpose. We find a total of 717 unique migration instances, spanning 175 repositories and 48 library pairs that we use for our experiments. The 48 library pairs include 24 unique source libraries and 37 unique target libraries, and 55 unique libraries in total. Across the migration instances, the minimum number of files with source library usages to migrate is 1 and the maximum is 183, with a mean of 4 and a median of 2 files to migrate per migration instance.

In our initial LLM empirical study (Section 3), we see that GPT-4o performs better than the other two models. Therefore, we use GPT-4o (gpt-4o-2024-11-20) to run MigrateLib in this evaluation.

5.2 Evaluation metrics

Recall from Section 4 that MigrateLib generates $\text{TestReport}_{premig}$ before the migration, and three test reports after each migration step: $\text{TestReport}_{llmmig}$ af-

ter the LLM migration step, $\text{TestReport}_{merge}$ after the merge skipped code, and $\text{TestReport}_{async}$ after the infer async transformation. For a given post-migration test report, we define the *correctness* of a migration after that step as the proportion of tests that pass in the test report compared to the tests that passed in the $\text{TestReport}_{premig}$. For example, if 80 tests passed in $\text{TestReport}_{premig}$, and 60 of *those* tests passed in $\text{TestReport}_{llmmig}$, then the correctness after the LLM migration step is 75% (60/80).

It is possible that some migrations may not be fully correct even after all the MigrateLib steps. We want to see how much migration work is still needed after MigrateLib migration to get a fully correct migration. Therefore, we randomly select one migration instance from each of the library pairs and attempt to manually fix them.

After fixing a migration as much as possible, we run the tests again and generate a new test report, which we refer to as $\mathit{TestReport}_{manual}$. We then compute the correctness of the manually fixed version. For the migrations which become fully correct, i.e., 100% correctness, we estimate the amount of manual work needed to fix them to answer RQ-2.2. We use line number of modified lines to estimate the amount of manual change required to complete a migration. For a migration, we compute a diff between the code before manual migration and the code after manual migration. From the diff, we calculate the number of changed lines of code by adding number of removed lines and number of added lines. We refer to this number as MigLOC_{manual}. Now, this number gives us the absolute number of lines changed, therefore we cannot directly compare this number across different migrations. For example, a migration that required 50 changed lines where the tool performed 48 of them and we only have to edit 2 lines should not be considered the same as a migration that required 4 lines of code and we still have to edit 2 lines. Therefore, we also measure the number of lines changed by MigrateLib during the automated migration from the diff between the code before automatic migration and the code after migration by MigrateLib. We refer to this number as MigLOC_{auto}. Accordingly, the total number of lines that needed to be changed for the migration is the sum of these two numbers. We use these to normalize the MigLOC_{manual} as follows:

$$\mathit{MigLOC}'_{manual} = \frac{\mathit{MigLOC}_{manual}}{\mathit{MigLOC}_{auto} + \mathit{MigLOC}_{manual}} \times 100 \tag{1}$$

Similarly, normalized automatic migration changes is calculated as the ratio of automatic changes to the total changes.

$$\mathit{MigLOC}'_{auto} = \frac{\mathit{MigLOC}_{auto}}{\mathit{MigLOC}_{auto} + \mathit{MigLOC}_{manual}} \times 100 \tag{2}$$

5.3 RQ-2.1: How many migrations can MigrateLib perform correctly?

Correctness according to applied steps.: Table 5 summarizes the results of RQ-2.1, while comparing the different migration steps. The table shows multiple metrics: the percentage of fully correct migrations is the proportion of migrations with 100% correctness. Median correctness is the median of the correctness value across all evaluated migrations. Number of migrations applied to is the number

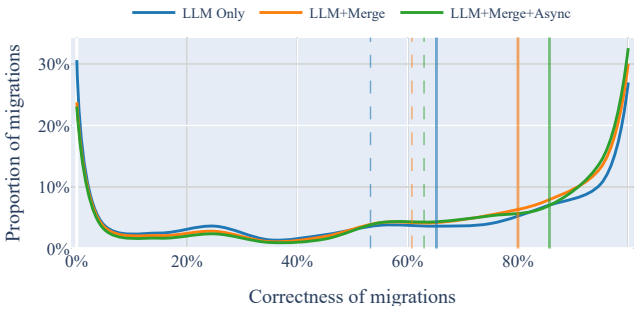


Fig. 9: Distribution of correctness of migrations.
The dashed lines indicate the mean, the solid lines indicate the median.

Table 5: Comparison between migration steps

Metric	LLM Only	LLM+Merge	LLM+Merge+Async
# migrations applied to	717	260	129
# (%) fully correct migrations	193 (27%)	215 (30%)	233 (32%)
Median correctness	65%	80%	86%
# (%) Migrations improved	–	88 (34%)	45 (35%)
# (%) Migrations remained same	–	131 (50%)	67 (52%)
# (%) Migrations got worse	–	41 (16%)	17 (13%)
Average improvement in correctness	–	46%	65%

of migrations where this step was applicable and thus got applied. Number of migrations improved shows the number of migrations where correctness improved in comparison to the previous step. Average improvement in correctness is the mean of improvement for the migrations where correctness improved after a post-processing step. Note that the post-processing steps are applied only when the migrations are not fully correct and when that specific post-processing step is applicable. For example, as shown in Table 5, merge skipped code is applied to 260 migrations, among which 88 (34%) migrations have improved correctness with an average improvement of 46% per migration. On the other hand, infer async transformation is applied to 129 migrations, among which 45 (35%) migrations have improved correctness with an average improvement of 65% per migration.

The frequency curve chart in Figure 9 shows the distribution of correctness of migrations after each migration step. All three distributions have concentrated values at 0% and 100%, indicating that most of the migrations either fail all tests or pass all tests. We see that the median correctness of migration increased from 65% in the LLM migration step to 80% after the merge skipped code, and further increased to 86% after the infer async transformation. Also note that out of the three migration steps, LLM-only has the highest proportion of migrations with 0% correctness; this proportion decreases after each post-processing step. Similarly, the proportion of migrations with 100% correctness increases after each post-processing step. The distributions show that after post-processing, there are fewer migrations with zero or low correctness, and more migrations with high correctness. This indicates that the post-processing steps are effective in improving the correctness of migrations.

To further understand when post-processing steps help or hinder the migration process, we manually analyze some of the migrations where the post-processing steps were applied. As shown in Table 5, merge skipped code got applied to 260 migrations, among which 88 (34%) migrations have improved correctness. The most common scenario for the improved correctness cases is that the LLM skipped code at the end of long files, classes, or methods. In most cases, these parts did not even contain any source library API usage, and therefore, the LLM did not need to change them. Accordingly, adding back these parts as is ensured that the migrated code is complete and does not miss any code that may be required for the tests to pass. We found one interesting case in a *requests→httpx* migration in the bheinzerling/bpemb repo. Here, the LLM removed function `_get_project` from the middle of the file. While this particular function did not require any migration, the LLM did edit parts of the file after the function was removed. Thus, the LLM did not simply stop migration like the common observed case; rather it intentionally removed this function. The LLM likely removed this function because it was not used within the file; however, the function was used in the test suite and thus its removal causes tests to fail. Our merge skipped code post-processing step added this function back to the migrated code, which improved the correctness of the migration. Interestingly, we found that the consequent commit in the repo added source code that uses this function, which indicates that the LLM's decision to remove this function was not correct, and our post-processing step helped to fix this mistake. As Table 5 shows, applying merge skipped code did not change correctness in 131 (50%) migrations. This was usually because the migrated code itself had multiple problems that led most of the test suite to fail, with none of the failures particularly related to the skipped code. Consequently, restoring the skipped code did not improve correctness. In 41 cases (16%), applying merge skipped code reduced correctness. Our manual analysis revealed that alignment does not always work as expected when the skipped code occurs between other code fragments that were migrated. Specifically, reinserting the skipped code into the migrated code at positions shifted by the migration sometimes introduced syntax errors, causing all tests to fail.

As Table 5 shows, infer async transformation got applied to 129 migrations, among which 45 (35%) migrations have improved correctness. In these cases, our qualitative analysis shows that MigrateLib correctly added the `async` or `await` keywords to/within the right functions, even when those functions appear in additional files that do not use the source library API and were accordingly not provided to the LLM. We find only 17 cases where applying infer async transformation made the correctness worse. All of these cases were due to very specific corner case usages of the usage of asynchronous functions in Python that MigrateLib does not currently handle. One example is when a function that got asynched during migration is called in a constructor of a class (i.e., __init__ method). For example, assume `fetch_year` from Figure 8 is called in a constructor of a class. Since MigrateLib identifies the constructor as a caller of the asynched function, it adds the `async` keyword to the constructor. However, asynchronous constructors are not allowed in Python, causing further tests to fail after the change. Instead, the expected workaround is to use a asynchronous factory method to create the object. There are only a few such corner cases, and handling them can be future feature enhancements to MigrateLib. In the remaining 67 (52%) cases, the infer async transformation did not change correctness, which was usually because the

migrated code itself had multiple problems that led most of the test suite to fail, with none of the failures particularly related to the async transformation.

Difficulty of attempted migrations.: To understand how difficult the migrations MigrateLib attempted are, we also examine the number of diff hunks per migration, number of changed lines per migration (total of number added plus deleted), and the program elements that appear in the changes. The number of hunks in a migration indicates how many separate code changes did MigrateLib make in the migration, while the number of changed lines indicates the size of the migration. We find that the minimum number of hunks in a migration is 1 and the maximum is 1,738, with a mean of 36 and a median of 14 hunks per migration. As for size, the minimum number of changed lines in a migration is 1 and the maximum is 5,644, with a mean of 217 and a median of 64 lines changed per migration.

Automatically determining all code changes according to PyMigTax is difficult as it goes beyond simple AST diffing. To still provide some proxy for the types of changes involved in the migration, we parse the ASTs to identify the program elements that appear in the changed lines of the migrations. We count the program elements that appear in the changed lines in both the before and after files. Since there are many AST node types, we focus on a few key program elements that are commonly involved in library migrations, including function calls, function definitions, exceptions, decorators, and attributes. We find that there are only 5 migrations where only an import replacement happened. In all remaining migrations, the changes involved actual code changes. Function call replacements happen in 97% of the migrations, with an average of 67 function call replacements per migration. Exceptions are the second most common type of change, occurring in 37% of the migrations, with an average of 2 exception changed per migration. Decorators and attributes also appear in the migration changes, occurring in 33% and 81% of the migrations, respectively, with an average of 9 and 28 changes per migration, respectively. Overall, this demonstrates that the migrations in our evaluation are not trivial, and involve a variety of code changes beyond just import replacement, which makes the migration process more complex and thus provides a good test for MigrateLib's capabilities.

Since we do not know the exact code changes required to complete each migration, we cannot directly compare the difficulty of the migrations that MigrateLib successfully completed vs. those it did not complete. The only pre-determined piece of information we have about the migrations is the number of files that require migration, which can be a proxy for the size of the migration. Figure 10 shows the achieved correctness vs. number of input files to migrate for each migration. We can see that migrations with a high number of input files can still achieve high correctness, and there is no strong correlation between the number of files to migrate and the achieved correctness.

For the 229 migrations that are fully correct, we have a "ground truth" of the required code changes to complete the migration. Thus, if we specifically look at these 229 fully correct migrations, we find that the number of files in those migrations ranges from 1 to 17, with average 2. The number of hunks ranges from 1 to 180, with average 13, and the number of changed lines ranges from 1 to 3,889, with average of 140. The code changes involved in these migrations are also not

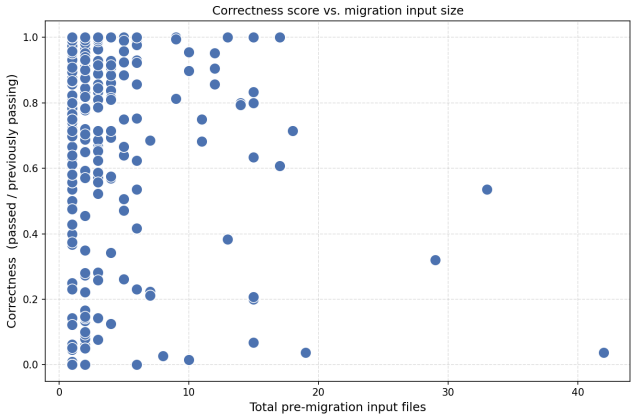


Fig. 10: Correctness vs. number of files to migrate. One outlier with 183 is excluded.

trivial, with 97% of them involving function call replacements, 29% involving exception changes, 19% involving decorator changes, and 73% involving attribute changes. This gives us further confidence that the migrations MigrateLib successfully completed are not trivial and vary in size and amount of required code changes.

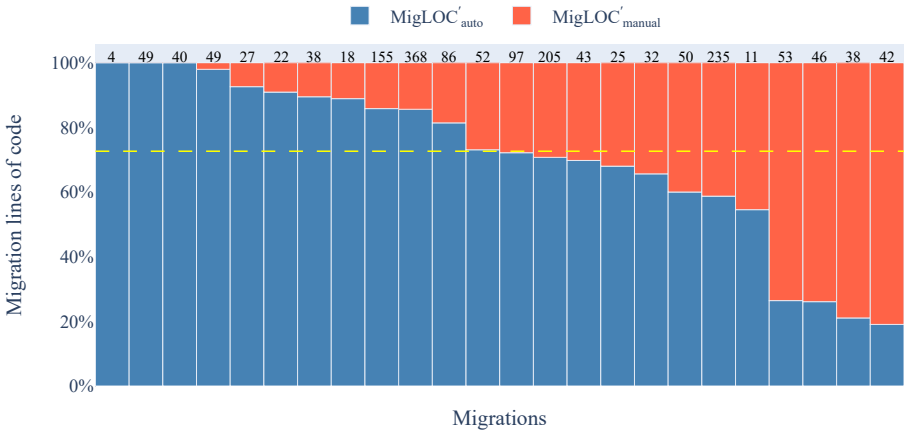
RQ-2.1

MigrateLib is able to fully migrate 32% of the migrations without any manual intervention. The median correctness of migrations improves from 65% with only the LLM migration step to 86% after the post-processing steps.

5.4 RQ-2.2: How much further code changes are required to complete the incomplete migrations?

RQ-2.1 shows that 233 migrations are completely correct after the automated migration process. The remaining 484 migrations will therefore require the developers to fix/complete them. To mimic what developers would do, we attempt to fix the migrations ourselves. However, the 484 migrations are between 43 unique library pairs in 147 repositories. To reduce the amount of manual work needed, while keeping the dataset diverse, we attempt to manually fix one incomplete migration from each library pair. Our selection results in a total of 43 migrations between 43 library pairs in 39 unique repositories, which we proceed to fix.

We are able to fix 24 out of 43 migrations. We could not fix the remaining 19 due to one or more of the following reasons: (1) As outsiders to the project, some of the application logic was too complex for us to understand and fix the migration. (2) In one case, the LLM migrated to *dataclasses* instead of *cattrs*, which made the migration incorrect. Fixing this migration would require undoing the entire migration and re-migrating it to the correct library, which is beyond the scope



The yellow dotted line shows median automatic change across all 24 migrations. The numbers on top of each bar show the total changed lines of code (automatic + manual) for that migration.

Fig. 11: Distribution of manual and automatic changes across manually fixed migrations.

of fixing an incomplete migration. (3) In some cases, the tests themselves need to be updated to account for the behavior of the new target library. In several cases, we could not figure the correct expectation in the test oracle (again due to the complexity of the application) or because the original tests had complex mocking of the source library APIs. (4) We did observe cases where additional dependent libraries require migration, beyond the source-target pair. We consider such migrations beyond scope of this work, thus we did not attempt to fix such migrations.

For the 24 migrations we were able to completely fix, we measure $MigLOC'_{manual}$ and $MigLOC'_{auto}$ using Equation 1 and Equation 2, respectively. Figure 11 shows the proportion of automatic and manual changes made for each migration. The blue part of the stacked bar is the $MigLOC'_{auto}$, while the red part is $MigLOC'_{manual}$. The yellow dotted line shows the median of $MigLOC'_{auto}$ across all 24 migrations, which is 73%. This means that for a median migration, MigrateLib performed 73% of the changes, while the developers would only need to perform 27% of the changes. Note that 3 bars in the figure have no red part, this is because the manual fixes required to successfully complete the migration were not source code changes but some environment change, such as installing a missing dependency.

RQ-2.2

For the migrations that required manual fixes, MigrateLib performed a median of 73% of the migration changes, while a developers would only need to perform 27% of the changes to complete the migration.

To understand where MigrateLib fell short, we qualitatively analyze the manual fixes we had to perform. We categorize them into the following observations. We provide (adapted) examples from the actual migrations.

- **The LLM often incorrectly uses target APIs:** Sometimes the target API call used by the LLM does not correctly replace the source API call. This includes incorrect import statements, incorrect arguments to API calls, missing API calls, incorrect exception types, incorrect use of `async`/`await`, missing non-default argument, and other API related mistakes. For example, in a *click→typer* migration, the LLM did not use a name for the `Typer()` call, which was required. The equivalent `Click` API does not require a name, so MigrateLib missed adding it.
- **The LLM skips part of the code:** Despite explicitly mentioning in the prompt not to skip parts of the code (Listing 2), the LLM still skips part of the code in migrations, especially when the code pattern is repetitive or does not have source API usage. We observe the same behavior in Section 3, which inspired us to add the *Merge skipped code* post-processing step in MigrateLib to merge the skipped code with the migrated code (see Section 4.3). However, the heuristics used in Merge skipped code still fail to identify some skipped code so we had to manually add it back in the fix.
- **MigrateLib struggles with complex async migrations:** Some sync to async migrations require complex changes in the code structure, such as changing class constructors to async methods, changing sync context managers to async context managers. The LLM often struggles to perform such complex changes correctly. Our infer async transformation in tool only performs simple transformations, such as adding `async`/`await` keywords and changing context managers. We manually fixed some of these complex async migrations.
- **The LLM refactors public program elements:** In the LLM migration step, MigrateLib migrates each file in isolation, which can lead to incorrect migrations. For example, In the *sqlalchemy→tortoise-orm* migration, MigrateLib renames a public class `EncryptedAlchemyBinaryField` defined in the client project to `EncryptedTortoiseBinaryField` to match the target library's name. This class is used in other parts of the code. Renaming the file prevents its callers from finding it, causing the errors in tests. Since MigrateLib does not pass the code that contains usage of the class in other files to the LLM, the LLM does not understand that it should not rename the class. Note that the prompt (Listing 2) does ask not to change any API names, however, the LLMs do not always strictly follow the instructions.
- **The LLM does not have the dependent code:** Figure 12 shows an example where the underlying LLM was not able to add an additional API call (`settings.reload()`) required for correct migration. This is required because the target library, *dynaconf*, loads the environment variables when the library is imported for the first time, and a reload is required if the environment variables are changed. The original code uses *python-dotenv* to load the environment variables, which reads the environment variables at runtime. Notice how the test, which is in another file that does not get passed to the LLM, updates the environment variables before running the tests. This updating happens after the library is imported, causing the tests to fail without the explicit `settings.reload()` call in the migrated code. If the LLM knew about

Original code

```
gh_actor = os.getenv("GH_ACTOR", "nb")
gh_app_id = get_int_env_var("GH_APP_ID")
```

MigrateLib migration

```
gh_actor = settings.get("GH_ACTOR", "nb")
gh_app_id = get_int_env_var("GH_APP_ID")
```

Manual fix

```
settings.reload()
gh_actor = settings.get("GH_ACTOR", "nb")
gh_app_id = get_int_env_var("GH_APP_ID")
```

Test code

```
@patch.dict(
  os.environ,
  {"GH_ACTOR": "testactor"}
)
def test_get_env_vars(self):
  expected_var = EnvVars(
    "testactor",
  )
  result = get_env_vars(True)
  actual = str(result)
  expected = str(expected_var)
  assert actual == expected
```

Fig. 12: Example of requiring to call an additional API in a *python-dotenv→dynaconf* migration.

the dependent code, it may be able to better understand the context of the migration.

- **The LLM is unaware of configuration requirements:** In some cases, the migration requires changes in the project configuration rather than code changes. For example, in a migration from *chardet→cchardet*, adjusting the Python version from `3.11` to `3.9` resolved the test failures. Similarly, we made a migration from *flask→sanic* successful by installing an additional dependency, *sanic-ext*, with no changes to the migrated code. The LLM is unaware of such configuration requirements, and therefore, it cannot perform such changes.
- **Often test expectations need to be updated:** There are cases where the tests require changes due to the target library's different behavior. For example, often the source library raises a generic exception, which the tests expect, while the target library raises a more specific exception. Another example is *coloroma* uses integers to represent colors (green is `32`), while *rich* uses strings (green is `"[green]"`).
- **MigrateLib does not handle mock migrations:** Some tests mock the source library's API calls in the tests, and these mocks need to be migrated to the target library's API calls. This is something we did not consider when designing MigrateLib, and therefore, it does not handle such cases.
- **MigrateLib does not support multi-library migrations:** We design MigrateLib to migrate from just one source library to just one target library, which may not always be sufficient. The project miguelgrinberg/microblog-api is such an example, which uses *flask* and some other libraries that depend on *flask*. To fully migrate to *fastapi* (or any other target library), we also need to migrate all these dependent libraries.

6 Discussion

Overall, we find that MigrateLib automatically completes migration for a large portion of cases (32% migrations) and does most of the required code changes (73% changes) for the remaining migrations, often leaving few edits for the developers

to perform (27% changes). While further improvements in the migration process may potentially help in reducing the manual effort, the remaining effort seems fairly minimal in most cases, specially for the developers who are familiar with the libraries and the project. We now discuss some potential improvement and future research and development directions for MigrateLib based on our findings, as well as its potential usage in practice.

6.1 Potential Improvements

A. Handling Refactorings: The two post-processing steps currently included in MigrateLib show that simple automated steps can significantly improve the correctness of migrations. However, Section Section 5.4 also discussed that that our current implementation has limitations and can therefore be further improved. For example, to handle problems introduced by refactorings, we can follow ideas from refactoring-aware merging (Ellis et al., 2022) where MigrateLib can detect (Tsantalis et al., 2022) and revert refactorings before applying the migration changes, and then re-apply the refactoring changes on the whole codebase.

B. Balancing Context and Re-prompting: We provide the LLM with a fairly straight-forward context, the file that uses the source library APIs. This is often insufficient, and on the contrary, it is often too much if the file is large but only a small part of it needs migration. Identifying and sending only relevant parts of the code may help the LLM to perform better migration. This however, comes with the cost of more pre-processing to identify the relevant code, as well as more post-processing to replace old code with the new code in the project.

Instead, another direction is to consider iteratively re-prompting the LLM with additional context, such as test error messages after running the tests on the migrated code. The challenge here is to identify meaningful error messages and generate a prompt that can help the LLM to understand the context better. Trying out different re-prompting strategies for this may be an interesting future research direction.

C. Handling Dependent Libraries: MigrateLib currently supports migrating from one source library to one corresponding target library. However, migrating from one library often requires also replacing related auxilary libraries. In Section 5.4, we did indeed observe some migrations that required additional dependent libraries to be migrated.

Since it is unrealistic to expect the developer to know all auxiliary libraries that need to be used with the target library, one potential way to handle this is to add a separate step before the migration where the LLM is instructed to check the existing dependency list and decide if other auxilary libraries need to be migrated as well. While this approach offloads the responsibility of identifying combined analagous libraries to the LLM, it also introduces additional complexity in managing and maintaining the migration process. The actual migration step will require changing the prompts and instructions to the LLM to handle multiple source and target libraries, which may be more challenging for the LLM. This is an interesting avenue for future research, likely requiring some experimentation to find the right balance between providing enough context and not overwhelming the

LLM with too much information. We do note though that our post-processing steps do not depend on the (number of) libraries involved and can therefore continue to be used as is in a multi-library scenario.

D. Adapting MigrateLib for Other Languages: While MigrateLib is built specifically for Python, its general approach can be applied to other programming languages as well. The language-specific parts of the tool relate to the pre- and post- processing steps, such as building the call graph and identifying skipped code, which can be easily adapted for other languages through some engineering effort. The LLM usage part itself is language-agnostic, with minimum changes required to the prompt to adjust to the input code's language.

6.2 Using MigrateLib in Practice

A. Test Coverage Requirement: MigrateLib depends on the existence of high-coverage tests in the project. While a developer is free to use MigrateLib on a project without tests, they need to be aware that the migration results may not be reliable. Specifically, without tests, the preprocessing step (Section 4.1) will miss identifying files that require migration but do not directly import the source library. After the LLM migration, MigrateLib will also not be able to validate the migration results due to the absences of tests. Finally, the post-processing steps will not run because there will be no difference between $\text{TestReport}_{premig}$ and $\text{TestReport}_{llmmig}$. With these limitations, a developer who uses MigrateLib on a project without tests will only perform the LLM migration, with no way to verify the migration. The developer will then need to verify the functionality themselves, outside MigrateLib's pipeline.

B. Using MigrateLib vs. a General-purpose Agent: AI coding agents, such as GitHub copilot, Cursor, Devstral, have recently gained attention for performing various software engineering tasks, including code transformation tasks(Yao et al., 2023). This naturally leads to the question of whether AI agents can be used to perform automated library migration, without the need for a specialized tool. The main advantage of AI agents is that they can autonomously perform the given task by iteratively deciding what code to change, what tests to run, and how to fix the migration based on the test results. In other words, using agents can potentially replace the pre- and post-processing steps we specifically developed for the library migration task. To understand the potential of using AI agents for library migration, we run a small experiment with 15 migration tasks from our dataset. To ensure diversity, we used stratified sampling to select migrations between 15 different library pairs with varying correctness levels of MigrateLib's migrations (1 migration with 0% correctness, 4 with $(0, 30\%]$, 4 with $(30, 60\%]$, 3 with $(60, 80\%]$, and 2 with $(80, 100\%]$). We use the Github Copilot Local agent with GPT-4o as the underlying LLM. We choose to use the same LLM as our experiments in Section 4 to have a fair comparison between the agentic and non-agentic approaches. We find that the agent achieved a higher correctness score for 6 out of the 15 migrations (median improvement of 27%), 7 were relatively worse (median 25% worse), and 2 were the same at 0% and 14% correctness. These results suggest that while AI agents can be helpful for library migration, they are

not necessarily better than a targeted LLM-based pipeline like MigrateLib. This aligns with the findings of Xia et al. (Xia et al., 2025) in the context of automatic program repair, where they find that an "agentless" approach through a targeted LLM-based pipeline can outperform agentic approaches.

C. Cost of using MigrateLib: The cost of using LLMs can be a concern for the developers. Therefore, we try to estimate the cost of using MigrateLib for library migration. The total number of tokens sent to the LLM across all migrations in our dataset is 3,834,385, and the total number of tokens received from the LLM is 2,491,712, with a total of 6,326,097 tokens. The median number of tokens sent to the LLM per migration is 2,704 and the median number of tokens received from the LLM per migration is 2,110. As of 2 June 2026, the cost of using GPT-4o is US$2.5 per million input tokens and US$10 per million output tokens. Based on these numbers, the estimated cost of using MigrateLib for migrating a median migration in our dataset is US$0.028. The migration in our dataset that consumed the most tokens has an estimated cost of US$0.38. This is a fairly low cost for the value that MigrateLib can provide in automating a large portion of the migration work.

7 Threats to validity

7.1 Internal validity

There were several manual steps involved in our evaluations. Manual activities are inherently subjective and may contain errors. In Phase 1, we manually validated the correctness of code changes and labeled any non-migration related change as refactoring or non-refactoring. To minimize subjectivity, two authors independently perform these steps and resolved disagreements through discussion. In Phase 3, only one author performs all manual migrations. However, we rely on objective test outcomes as the primary measure of correctness.

All the migrations in PyMigBench happened prior to the known training cutoff date of the three models (Late 2023). Therefore, there is a possibility that the LLMs are simply reciting the migrated code they have seen before (for this particular code base). This means that the reported results in Phase 1 may be an upper bound or optimistic estimate of the LLMs' migration capabilities. While this setup was essential in Phase 1 to allow us to compare the LLM migrations to a developer-performed ground truth migration for systematic evaluation and to identify the types of migration-related code changes LLMs excel at vs. struggle with, we designed a different evaluation approach in Phase 3 to overcome this threat, and thus provide a more realistic estimate of the LLMs' migration capabilities. To further quantify the extent of data leakage in phase 3, we search the history of the repositories we use to see if the target library ever appeared there (e.g., the reverse migration happened before). We find that only 54/717 (7.5%) migrations had the target library in their dependencies before. While not absolutely guaranteed since we do not have access to the training data, this strongly suggests that the majority of our evaluation is on code that the LLM has not seen the target library being used for before.

The merge skipped code uses some thresholds to identify skipped code hunks, which we determine through experimentation. Different thresholds may yield different results. However, we believe our chosen thresholds strike a reasonable balance between adding back genuinely skipped code and avoiding incorrect additions. If the assumptions do not hold, the resulting migration may potentially fail, therefore, the results we have may be a lower bound of the actual migration correctness.

7.2 External validity

We use three off the shelf LLMs in Phase 1, and the best of them in Phase 3. Using an LLM specifically tuned for the library migration task could potentially yield better results. Therefore, our findings in this paper may be viewed as a lower bound of the performance of potentially more powerful and specialized models.

In Phase 3, we focus on Python projects that are well-tested and actively maintained, as selected through specific filters. This focus may limit the generalizability of our findings to projects with less test coverage, different maintenance status, or other programming languages. The set of library pairs and migration scenarios is also limited by the availability of analogous libraries and the manual validation process. As a result, the observed effectiveness of MigrateLib may not fully generalize to all migration contexts. That said, the size of the dataset (717 migrations) and the diversity of selected projects (175 repositories) provide a reasonable basis for evaluating the tool's effectiveness in a variety of migration scenarios.

The libraries we consider in Phase 1 and Phase 3 are relatively popular and well-established. This familiarity with APIs and API usage is precisely our hope when using LLMs for migration. However, being familiar with the APIs alone does not always guarantee being able to map them to each other or apply the right transformations on all code bases, which is why our evaluation is necessary. For newer libraries, additional information, e.g., API documentation, may be needed to guide the LLM in the migration process. This can be an interesting avenue for future study, and thus improvement of MigrateLib.

7.3 Construct validity

In RQ-1.1, we analyze the code change categories based on Change_{dev}, not on Change_{llm}. The categories can differ if the LLM produces a different code change than the developer. PyMigBench already provides us with the Change_{dev} categories; categorizing the LLM changes according to PyMigTax (Islam et al., 2024) requires additional manual effort that is beyond the scope of this work. That said, our observation is that the code/API differences in most cases are minor, and would not yield in a different PyMigTax category.

We use automated test suites to assess migration correctness in Phase 3, but these tests may not cover all possible behaviors or edge cases in the migrated projects. To minimize this, we select projects with at least 80% lines of code covered by tests, which helps ensure that the tests are comprehensive enough to provide a reliable measure of correctness.

We use migration lines of code (Equation 1) to estimate the amount of migration-related code changes left for the developer to make. This metric only captures the

quantity of changes and does not capture all aspects of migration work, such as time spent understanding and comparing the source and target APIs, or debugging the migration changes. We choose to use this metric since it is objective and measurable. We additionally accompany it with a qualitative analysis of the types of changes we had to make, which can provide more insights into the nature of the remaining work.

8 Related Work

Prior work studied how developers compare analogous libraries (Larios Vargas et al., 2020; de la Mora and Nadi, 2018; Tanzil et al., 2024) as well as the factors that lead them to decide to replace one library with another (He et al., 2021). Such reasons include project integration needs, lack of maintenance of the current library, better usability or features of the new library, and dependency simplification. In contrast, most library migration techniques (discussed below) start their work with the assumption that the target library has already been identified and focus on automating the migration process itself. This is also our focus in this paper.

The majority of the existing library migration techniques focus on identifying API mappings between two analogous libraries. Some of these techniques mine existing library migrations to identify the API mappings (Teyton et al., 2012, 2013; Alrubaye et al., 2019b,a, 2020). Some techniques see the requirement of existing migrations as a limitation, and therefore use other techniques, such as machine learning (Alrubaye and Mkaouer, 2018; Chen et al., 2019) and natural language processing (Ni et al., 2021). These techniques only identify the API mappings, but do not perform client code transformation, which is a separate challenge.

SOAR (Ni et al., 2021) uses API documentation to identify API mappings, and then uses the mappings and unit tests to guide program synthesis for client code transformation. SOAR has several limitations that prevent it from being a general solution for library migration. SOAR is evaluated on only two pairs of deep learning libraries, focusing on their migration of neural network models. These APIs are usually called in a sequence, where the output of one call is used as input to the next call. The nature of these APIs allows SOAR to check the program state after each API call, which is not the case for most libraries. Additionally, given the limited number of libraries, their technique relies on the library-specific error message format to further guide the synthesis process, requiring extensive changes and technique adaptations for running SOAR on other libraries. SOAR supports one-to-many code changes for a small set of pre-determined APIs, whose information is hard-coded for the transformation process. SOAR also times out for 20% of the evaluated migrations, even for small code snippets, the largest being 183 lines of code.

Zhou et al. (Zhou et al., 2023b) proposed HaPiM, which first trains an unsupervised machine learning model capable of API mapping, and then uses the API mappings to guide an LLM in code transformation. Similar to Chen et al. (Chen et al., 2019), HaPiM learns from API usage patterns of source and target libraries in their respective client projects, therefore does not require any existing migration examples. For the code transformation, they use the API mappings to prompt an LLM to replace source API calls with target API calls. Their approach

outperforms MigrationMapper (Alrubaye et al., 2019b) and GPT-3.5 Turbo on the BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020) metrics evaluated on 5 Java library pairs. A significant limitation of HaPiM is that it works with small independent code snippets that use the source library APIs, and results in an equivalent snippet that uses the target library APIs. This is as opposed to real-life code where the source API usages are interleaved with other types of code, such as variable declarations, control statements, and other library usages. This means that to use HaPiM, developers need to first extract code snippets that contains only the source library API usages from the client code, then use HaPiM to replace them with target library APIs, and then re-integrate the transformed code back into the client code, which is not practical for large-scale library migrations.

Nikolov et al. (Nikolov et al., 2025) describe Google's experiences in using LLMs for several internal code transformation tasks, including *Joda Time→Java Time* migrations. They do not attempt to provide a general solution for library migration; rather, they use LLMs to perform the migration they need. Therefore, their prompts contains instructions and contextual information tailored specifically for *Joda Time→Java Time* migrations. Nonetheless, they show that using LLMs reduced migration time by at least 50% compared to manual migration.

There are also a few studies that focus on migrating between different versions of the *same library*. The study of Nikolov et al. (Nikolov et al., 2025) we describe above also includes *JUnit 3→JUnit 4* migrations. Almeida et al. (Almeida et al., 2024) evaluate the performance of ChatGPT for *SQAlchemy 1→SQLAlchemy 2* migrations. While similar, migrating between different versions of the same library is different from migrating between different libraries, because the former can leverage the evolution or release history of the libraries to identify API changes.

The above shows that there have been different approaches to automate the library migration process. However, these techniques come with one or more limitations that prevent them from being a general solution for library migration. Our tool, MigrateLib, resolves these limitations and can automate the entire library migration process for any arbitrary analogous library pair. We evaluate MigrateLib on a large and diverse dataset of 717 migrations between 48 analogous library pairs, and show that MigrateLib can successfully perform most of the migration with no or minimal manual intervention from the developer.

9 Conclusion

Our goal in this paper is to fully automate the Python library migration process. We systematically approached this goal by first conducting an empirical study to understand the capabilities of LLMs for automating Python library migration. Overall, we find that LLMs can handle complex code changes but we also identify some challenges that need to be addressed to make LLM-based migration practical. Based on these findings and insights, we built MigrateLib, a practical CLI tool that combines LLMs with static and dynamic program analysis to automate the full migration process. Our evaluation on 717 simulated migrations across 175 diverse projects and 48 library pairs shows that MigrateLib can fully automate a large portion of migrations (32%), with most remaining cases requiring only minor manual intervention (median 27% changes). The tool's layered approach, including post-processing steps, improves LLM migration correctness (from 65%

to 86%) and reduces developer effort. Overall, MigrateLib lowers the barrier for large-scale library migration in real-world projects, demonstrating accessible and effective automated library migration.

Acknowledgements We thank Dr. Ildar Akhmetov, Northeastern University, for his contributions to the early ideas of this work.

Declarations

Funding: This work was supported by the Canada Research Chairs Program and NSERC CREATE.

Author contributions: Below are the contributions of each author –

- *Mohayeminul Islam* contributed to idea conceptualization, design and implementation of the MigrateLib, conducting the experiments, writing script for the experiments, manual evaluations in Phase 1, manual migration fix in Phase 3, and writing the manuscript.
- *Ajay Kumar Jha* contributed to idea conceptualization, design of the MigrateLib, manual evaluations in Phase 1, and writing the manuscript.
- *May Mahmoud* manual evaluations in Phase 1, and revising the manuscript.
- *Sarah Nadi* contributed to idea conceptualization, design of the MigrateLib, manual evaluations in Phase 1, and writing the manuscript. She is the Principal Investigator of the project.

Data availability: The source code of MigrateLib, the experimental data, and the scripts and the detailed results of our experiments are available at `https://doi.org/10.6084/m9.figshare.29602517`.

Conflict of interest: No conflict.

Ethical approval/informed concent/clinical trial number: Not applicable.

References

Almeida A, Xavier L, Valente MT (2024) Automatic library migration using large language models: First results. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp 427–433

Alrubaye H, Mkaouer MW (2018) Automating the detection of third-party java library migration at the function level. In: CASCON, pp 60–71

Alrubaye H, Mkaouer MW, Ouni A (2019a) Migrationminer: An automated detection tool of third-party java library migration at the method level. In: 2019 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 414–417

Alrubaye H, Mkaouer MW, Ouni A (2019b) 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, pp 347–357

Alrubaye H, Mkaouer MW, Khokhlov I, Reznik L, Ouni A, Mcgoff J (2020) Learning to recommend third-party library migration opportunities at the api level. Applied Soft Computing 90:106140

Atil B, Chittams A, Fu L, Ture F, Xu L, Baldwin B (2024) Llm stability: A detailed analysis with some surprises. URL https://arxiv.org/abs/2408.04667, 2408.04667

Chen C, Xing Z, Liu Y, Xiong KOL (2019) Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. IEEE Transactions on Software Engineering 47(3):432–447

Cohen J (1960) A coefficient of agreement for nominal scales. Educational and psychological measurement 20(1):37–46

contributors TL (2025) Libcst. URL https://libcst.readthedocs.io

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

Di Rocco J, Nguyen PT, Di Sipio C, Rubei R, Di Ruscio D, Di Penta M (2025) Deepmig: A transformer-based approach to support coupled library and code migrations. Information and Software Technology 177:107588

Dubey A, Jauhri A, Pandey A, Kadian A, Al-Dahle A, Letman A, Mathur A, Schelten A, Yang A, Fan A, et al. (2024) The llama 3 herd of models. arXiv preprint arXiv:240721783

Ellis M, Nadi S, Dig D (2022) Operation-based refactoring-aware merging: An empirical evaluation. IEEE Transactions on Software Engineering 49(4):2698–2721

Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, Zhang JM (2023) Large language models for software engineering: Survey and open problems. arXiv preprint arXiv:231003533

Flask (2024) https://flask-socketio.readthedocs.io/en/latest/api.html

He H, He R, Gu H, Zhou M (2021) 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, pp 478–490

Huang Z, Chen J, Jiang J, Liang Y, You H, Li F (2024) Mapping apis in dynamic-typed programs by leveraging transfer learning. ACM Trans Softw Eng Methodol DOI 10.1145/3641848, URL https://doi.org/10.1145/3641848, just Accepted

Islam M, Jha AK, Nadi S, Akhmetov I (2023) Pymigbench: A benchmark for python library migration. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), IEEE, pp 511–515, DOI 10.1109/MSR59073.2023.00075

Islam M, Jha AK, Akhmetov I, Nadi S (2024) Characterizing python library migrations. In: Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE), DOI 10.1145/3643731

Islam MM, Jha AK, Mahmoud M, Akhmetov I, Nadi S (2025) An empirical study of python library migration using large language models. In: Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025)

Jeantine-Glenn W (2025) johnnydep. URL `https://pypi.org/project/johnnydep`

Katz J (2020) Libraries.io open source repository and dependency metadata. https://zenodo.org/record/808272, DOI 10.5281/ZENODO.808272

Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empirical Software Engineering 23(1):384–417

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. biometrics pp 159–174

Larios Vargas E, Aniche M, Treude C, Bruntink M, Gousios G (2020) 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, pp 245–256

de la Mora FL, Nadi S (2018) An empirical study of metric-based comparisons of software libraries. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), Association for Computing Machinery, DOI 10.1145/3273934.3273937

Murali V, Maddila C, Ahmad I, Bolin M, Cheng D, Ghorbani N, Fernandez R, Nagappan N (2023) Codecompose: A large-scale industrial deployment of ai-assisted code authoring. arXiv preprint arXiv:230512050

Nam D, Macvean A, Hellendoorn V, Vasilescu B, Myers B (2024) Using an llm to help with code understanding. In: 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), IEEE Computer Society, pp 881–881

Nguyen N, Nadi S (2022) An empirical evaluation of github copilot's code suggestions. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp 1–5

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

Nikolov S, Codecasa D, Sjovall A, Tabachnyk M, Chandra S, Taneja S, Ziftci C (2025) How is google using ai for internal code migrations? arXiv preprint arXiv:250106972

OpenAI (2024) OpenAI models. https://platform.openai.com/docs/models

Ozkaya I (2023) Application of large language models to software engineering tasks: Opportunities, risks, and implications. IEEE Software 40(3):4–8

Pallets (2024) click. https://pypi.org/project/click

Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp 311–318

Peng S, Kalliamvakou E, Cihon P, Demirer M (2023) The impact of ai on developer productivity: Evidence from github copilot. arXiv preprint arXiv:230206590

Ren S, Guo D, Lu S, Zhou L, Liu S, Tang D, Sundaresan N, Zhou M, Blanco A, Ma S (2020) Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:200910297

Tanzil MH, Uddin G, Barcomb A (2024) How do people decide?: A model for software library selection. In: 2024 IEEE/ACM 17th International Conference

on Cooperative and Human Aspects of Software Engineering (CHASE), ACM, DOI 10.1145/3641822.3641865

Teyton C, Falleri JR, Blanc X (2012) Mining library migration graphs. In: 2012 19th Working Conference on Reverse Engineering, IEEE, pp 289–298

Teyton C, Falleri JR, Blanc X (2013) Automatic discovery of function mappings between similar libraries. In: 2013 20th Working Conference on Reverse Engineering (WCRE), IEEE, pp 192–201

Tsantalis N, Ketkar A, Dig D (2022) Refactoringminer 2.0. IEEE Transactions on Software Engineering 48(3):930–950, DOI 10.1109/TSE.2020.3007722

Wang C, Jiang J, Li J, Xiong Y, Luo X, Zhang L, Hu Z (2016) Transforming Programs between APIs with Many-to-Many Mappings. In: Krishnamurthi S, Lerner BS (eds) 30th European Conference on Object-Oriented Programming (ECOOP 2016), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Leibniz International Proceedings in Informatics (LIPIcs), vol 56, pp 25:1–25:26, DOI 10.4230/LIPIcs.ECOOP.2016.25, URL http://drops.dagstuhl.de/opus/volltexte/2016/6119

Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q (2024) Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering

Wei Y, Xia CS, Zhang L (2023) Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 172–184

Xia CS, Wei Y, Zhang L (2023) Automated program repair in the era of large pre-trained language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, pp 1482–1494

Xia CS, Deng Y, Dunn S, Zhang L (2025) Demystifying llm-based software engineering agents. Proceedings of the ACM on Software Engineering 2(FSE):801–824

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

Yao S, Zhao J, Yu D, Du N, Shafran I, Narasimhan K, Cao Y (2023) React: Synergizing reasoning and acting in language models. DOI 10.48550/arXiv.2210.03629, URL https://arxiv.org/abs/2210.03629, 2210.03629

Zhang Z, Pan M, Zhang T, Zhou X, Li X (2020) Deep-diving into documentation to develop improved java-to-swift api mapping. In: Proceedings of the 28th International Conference on Program Comprehension, pp 106–116

Zhou B, Wang X, Xu S, Yao Y, Pan M, Xu F, Ma X (2023a) Hybrid api migration: A marriage of small api mapping models and large language models. In: Proceedings of the 14th Asia-Pacific Symposium on Internetware, pp 12–21

Zhou B, Wang X, Xu S, Yao Y, Pan M, Xu F, Ma X (2023b) Hybrid api migration: A marriage of small api mapping models and large language models. In: Proceedings of the 14th Asia-Pacific Symposium on Internetware, pp 12–21