An Empirical Study of Python Library Migration Using Large Language Models

Mohayeminul Islam
University of Alberta
Canada
mohayemin@ualberta.ca

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

May Mahmoud
New York University Abu Dhabi
United Arab Emirates
m.mahmoud@nyu.edu

Ildar Akhmetov
Northeastern University
Canada
i.akhmetov@northeastern.edu

Sarah Nadi
New York University Abu Dhabi
United Arab Emirates
sarah.nadi@nyu.edu

***Abstract*—Library migration is the process of replacing one library with another library that provides similar functionality. Manual library migration is time consuming and error prone, as it requires developers to understand the APIs of both libraries, map them, and perform the necessary code transformations. Large Language Models (LLMs) are shown to be effective at generating and transforming code as well as finding similar code, which are necessary upstream tasks for library migration. Such capabilities suggest that LLMs may be suitable for library migration. Accordingly, this paper investigates the effectiveness of LLMs for migration between Python libraries. We evaluate three LLMs, LLama 3.1, GPT-4o mini, and GPT-4o on PyMigBench, where we migrate 321 real-world library migrations that include 2,989 migration-related code changes. To measure correctness, we (1) compare the LLM's migrated code with the developers' migrated code in the benchmark and (2) run the unit tests available in the client repositories. We find that LLama 3.1, GPT-4o mini, and GPT-4o correctly migrate 89%, 89%, and 94% of the migration-related code changes, respectively. We also find that 36%, 52% and 64% of the LLama 3.1, GPT-4o mini, and GPT-4o migrations pass the same tests that passed in the developer's migration. To ensure the LLMs are not reciting the migrations, we also evaluate them on 10 new repositories where the migration never happened. Overall, our results suggest that LLMs can be effective in migrating code between libraries, but we also identify some open challenges.**

I. Introduction

Software libraries are essential for modern software development, as they provide reusable code that can significantly reduce development time and effort. Developers often replace one library with another in their applications to improve performance, address security vulnerabilities, or even for license compatibility [1, 2]. This process, known as *library migration* [3], is time consuming and error prone [2], as it requires developers to understand the Application Programming Interfaces (APIs) of both libraries, find API replacements (*API mapping*), and perform various code transformations [4].

There have been previous efforts towards automating library migration, but fully usable tools for this task are limited. With the exception of a few attempts [5–7], most of the existing library migration techniques only focus on finding API mappings[4, 8–11], without supporting the code transformation.

Large Language Models (LLMs) have shown that they can be effective in various software engineering tasks [12–14], including the necessary upstream tasks for library migration, such as code generation [15–17], code comprehension [18], and code transformation [19–21]. While this suggests that LLMs may be capable of performing library migration, we still need further empirical evidence to support this claim. Accordingly, this paper investigates the effectiveness of LLMs for library migration for a wide range of libraries and applications.

Our investigation focuses on Python, and we use PyMigBench [22], an existing dataset of 321 real-world library migrations containing 2,989 migration-related code changes. The data includes the code before and after the migration, along with detailed descriptions of the migration-related code changes including labeling of the types of changes using an existing taxonomy PyMigTax [23].

We use three state-of-the-art LLMs: LLama [24], GPT-4o mini [25], and GPT-4o [26] to perform the migrations. We refer to these models as LLama, Mini, and 4o in the rest of the paper, respectively. We evaluate the correctness of the migrations in two ways. First, we compare the LLM-migrated code with the developer-migrated code and report correctness at two granularity levels: the full migration and the individual code changes. We use PyMigTax [23] to identify which types of code changes the LLMs can/cannot handle. Second, for the subset of migrations with available unit tests, we run the available tests to assess the correctness of the migrated code.

We find that 4o performs the best among the three LLMs, while also being the most expensive. It correctly migrates at least one migration-related code change for 94% of the migrations and fully migrates 57% of the migrations. At the code change level, 4o correctly migrates 94% of the migration-related code changes. In the test-based evaluation, 64% of the migrations done by GPT-4o pass the same set of tests that passed in the developer's migration. Mini, the lower cost alternative of 4o, closely follows 4o in performance and

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00077

LLama performs comparatively worst.

Our analysis of the code changes using PyMigTax reveals that all three LLMs can perform reasonably well on the majority of the code change types. Specifically, their performance on higher cardinality code change, i.e., changes involving multiple APIs, is promising (70%, 79% and 84% correct). However, all three LLMs struggle with code changes that require argument transformation, including changes to the argument value or type. 4o outperforms the other two models in most of the code change types, especially in function call and higher cardinality code changes.

The evaluation setup above has the advantage of comparing to a developer-performed ground truth migration. However, it uses code that the models may have potentially seen during training, which poses a risk of reciting the migrated code. To address this, we conduct an additional experiment on 10 recent well-tested repositories using migrations that never appeared in their commit history. Overall, we find that even on unseen target code, LLama, Mini, and 4o perfectly migrate 10%, 40%, and 50% of the migrations respectively, which shows similar relative performance to the previous experiment.

Overall, our study reveals various new opportunities and challenges of using LLMs for library migration. We find that all three LLMs have high overall correctness, indicating the potential to play a critical role in reducing library migration manual effort. Also, LLMs are capable of transforming changes that were typically considered difficult, e.g., many-to-many code changes and changes between APIs having completely different styles. Challenges include all three LLMs struggling to handle some unit conversions, doing occasional unwanted extra changes, and generalizing exceptions. Practitioners can use these insights to decide if they want to use LLMs for library migration. Tool builders can build library migration tools around LLMs by addressing the limitations identified in this study, potentially combining LLMs with traditional program analysis techniques. The artifact for this study is available at https://figshare.com/articles/conference_contribution/25459000.

To summarize, our contributions in this paper are as follows:

1) We investigate the effectiveness of three LLMs for library migration using an existing dataset of 321 real-world library migrations, PyMigBench.
2) We evaluate migration correctness in two ways: (1) comparing it to the developer changes at both migration and code change levels and (2) using unit tests, which provides a run-time evaluation of the migrated code.
3) Using PyMigTax [23], we identify which types of code changes the LLMs can/cannot handle.
4) We verify LLMs' performance on unseen migrations.

II. Background and Terminology

A. Library Migration

Library migration is the process of updating a software project to replace a used library with another one that provides similar functionality [3]. The *source* library is the one being replaced, and the *target* library is the one that replaces it [3]. One *migration* instance refers to a commit in a repository where a migration happened from a specific source library to a target library [22], 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 [22]. 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 code. Change$_{dev}$ and Change$_{llm}$ denote code changes by developers and an LLM, respectively.

B. PyMigBench and PyMigTax

PyMigBench is a dataset of real-world Python library migrations [22], mined from version control history. We use the latest available version, version 2.2.5 [23] in our experiments. 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 [23], built using the first version of PyMigBench. The authors 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 annotated with the PyMigTax categories. We use PyMigTax categories (shown in Table III) to understand the types of code changes LLMs can handle. The PyMigTax paper [23] has full category descriptions.

III. Experiment Setup

A. Models

We use the latest available versions of the three LLMs at the time of experiment: Meta LLama 3.1-70B-Instruct, OpenAI GPT-4o mini-2024-07-18, and OpenAI GPT-4o-2024-08-06. LLama is free and open source with an input+output limit of 8,192 tokens and is trained up until December 2023 [24, 27]. The OpenAI models are proprietary, have a output token limit of 16,384 and are trained on data up until October 2023 [28].

B. Data Preparation

Two repositories from PyMigBench are no longer public, preventing us from using 2 migrations. We clone the remaining repositories containing 319 migrations. For each migration commit, we use PyDriller [29] to extract the content of each file that has recorded migration-related code changes, both before (File$_{pre}$) and after the migration (File$_{dev}$). We use Python's built-in `ast.parse` function [30] to ensure that both File$_{pre}$ and File$_{dev}$ are syntactically valid code and discard 5 migrations having files with syntax errors. We conduct our experiments with the remaining 314 migrations containing 2,910 migration-related code changes.

C. Migration

For each file having code changes, we use the LLM prompt template in Figure 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 [31], we use a temperature of 0 and run each migration 10 times for each model. We calculate the difference between the runs to understand the variability in the generated code. We use git-diff [32] 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 model to evaluate the LLMs' results. Our artifact contains the LLM's migrations for all 10 runs.

D. Migration Evaluation

Our goal is to assess the ability of LLMs to migrate Python code between analogous libraries. However, given that there may be more than one way to correctly migrate an API usage [33], the LLM's migrations might not exactly match the developer changes in PyMigBench, yet still be correct. Considering this, we use different strategies to assess correctness through the following research questions.

RQ1 **How similar are the LLM migrations to the benchmark migrations?** We consider Change$_{dev}$ 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.

RQ2 **How many migrations pass unit tests?** To evaluate runtime correctness of the migrated code, we use a second evaluation strategy where we run any available unit tests.

RQ3 **Can LLMs perform migrations they have not seen before?** We evaluate on 10 additional well-tested repositories that were updated after the models' training dates, using migrations that never happened in their version-control history. We run the tests to evaluate correctness.

> 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:

Fig. 1: LLM Migration Prompt

IV. RQ1 How similar are the LLM migrations to the benchmark migrations?

A. Approach

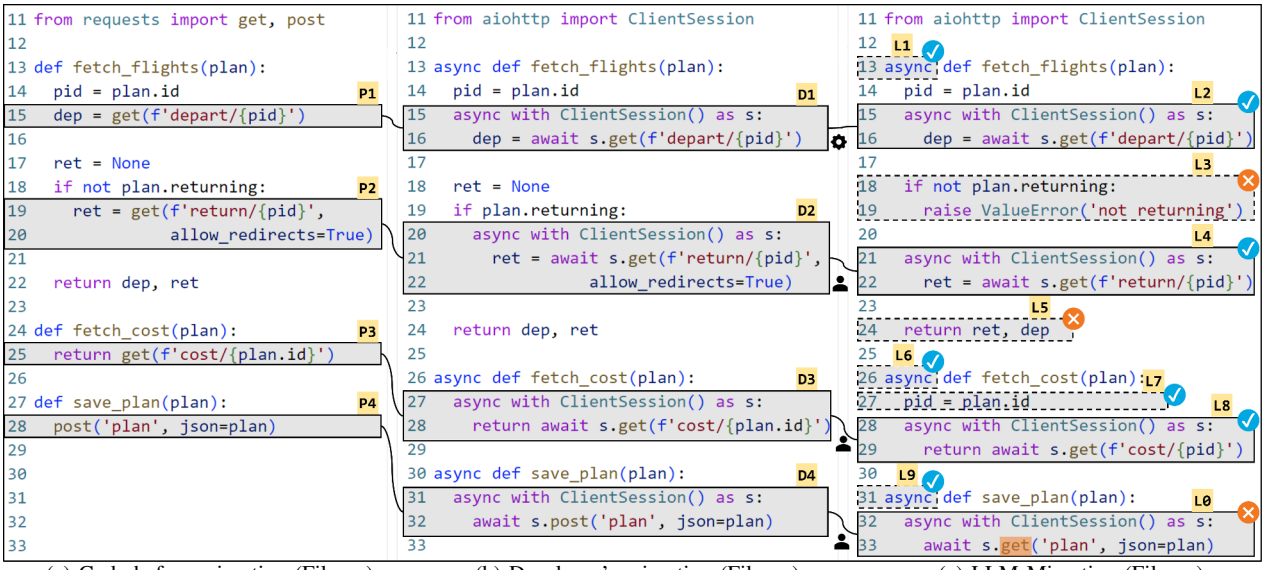
In this RQ, we assess each LLM's migration correctness by comparing its code changes (Change$_{llm}$) to those made by developers (Change$_{dev}$), while manually judging potential alternative changes. A migration may involve multiple files, each with multiple migration-related code changes. We evaluate each code change individually, and aggregate the results to determine the correctness of the migration. We now explain the process using the example in Figure 2 that shows a migration from the library *requests* to *aiohttp*.

1) Match code changes: The goal of this step is to compare the code changes the LLM made (Change$_{llm}$) with the developer ground truth changes (Change$_{dev}$), as well as manually finding alternative correct changes.

a) Auto change matching: We first attempt to automatically match as many changes as possible while ensuring precision i.e. not falsely marking a 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` [30].

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) Manual review: The manual review step focuses on reviewing uChange$_{dev}$ and uChange$_{llm}$. We use a three-way diff viewer to manually compare File$_{pre}$, File$_{dev}$, and File$_{llm}$ for each file with at least one uChange$_{dev}$ or 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



(a) Code 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 migration-related 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.

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 a refactoring. 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.

To ensure the quality of the manual review, we have two authors independently review changes. We first review 194 code changes from one model, and measure the agreement using Cohen's Kappa [34], but we do not reach substantial agreement. Therefore, we discuss and resolve disagreements and update our coding guideline. We then review another 114 code changes, and achieve a Cohen's Kappa score of 0.83 (almost perfect) and 0.73 (substantial) [35]. Accordingly, we proceed with only one reviewer for the remaining changes.

2) Determine migration 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 (`D1`–`L2`), or with an alternative API (`D2`–`L4`), or with same APIs but with some refactoring (`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.

Based on the individual matched code changes, we now automatically determine an overall migration's 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.

TABLE I: Correctness of PyMigBench migrations (RQ1, migration level)

Status	Number (percentage) of migrations		
	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%)

B. Findings: Migration level correctness

Table I 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%).

RQ1, 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.

C. Findings: code change level correctness

We now present the correctness results at the individual code change level. 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 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.

1) Overall code change correctness: Table II 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.

2) Code change correctness by category: To understand if some types of code changes are more difficult to migrate, Table III shows the distribution of code change correctness across the different PyMigTax categories. The second column shows how often each PyMigTax category appears in the 2,989 code changes in PyMigBench, which provides perspective on the category's frequency in practice. 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.

TABLE II: Correctness of PyMigBench migration-related code changes (RQ1, overall code change level)

Status	Number (percentage) of code changes		
	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

a) Program elements: Function calls are the most common program elements in the PyMigBench (60% frequency). 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* 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

TABLE III: Correctness of code changes across PyMigTax categories (RQ1, code change level by category).

PyMigTax category	Frequency in PyMigBench	LLama 3.1		GPT-4o mini		GPT-4o	
		#Eval	%Cor	#Eval	%Cor	#Eval	%Cor
Program elements							
Function call	1,804 (60%)	518	88%	1,093	87%	1,092	93%
Import	732 (24%)	276	95%	580	94%	593	98%
Decorator	411 (14%)	197	87%	335	90%	353	87%
Attribute	183 (6.1%)	63	83%	103	78%	103	89%
Type	55 (1.8%)	12	75%	34	85%	34	88%
Exception	29 (1.0%)	17	65%	29	86%	29	93%
Function reference	12 (0.4%)	3	33%	12	58%	12	100%
Properties							
No properties	262 (8.8%)	43	84%	150	88%	150	97%
Elmnt name change	1,025 (34%)	398	84%	773	85%	771	90%
Arg addition	578 (19%)	150	85%	244	80%	244	88%
Arg deletion	349 (12%)	175	85%	234	87%	234	89%
Arg transform	317 (11%)	115	77%	201	76%	220	77%
Param addition	127 (4.2%)	110	87%	127	95%	127	90%
Arg name change	112 (3.7%)	20	60%	58	59%	58	84%
Async transform	50 (1.7%)	35	83%	50	90%	50	92%
Output transform	49 (1.6%)	25	88%	49	78%	49	92%
Cardinality							
One-to-One	1,552 (52%)	408	86%	877	84%	894	91%
One-to-Zero	297 (10%)	99	99%	290	99%	290	100%
Zero-to-One	198 (6.6%)	34	97%	55	87%	55	89%
Higher Cardinality	*210 (7.0%)*	*94*	*70%*	*164*	*79%*	*164*	*84%*
One-to-Many	142 (4.8%)	61	69%	103	78%	103	80%
Many-to-One	41 (1.4%)	16	62%	34	82%	34	91%
Many-to-Many	27 (0.9%)	17	82%	27	78%	27	89%
Total	2,989	911	89%	1,966	89%	1,996	94%

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. We discuss specific failure instances in section VII.

The property *parameter addition to decorated function* is found in migrations from or to the click library [36], where some decorators require adding a parameter to the decorated function. 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 a target needs to be added, but no source needs to be removed. 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 [9, 11, 37, 38], the PyMigBench dataset shows that higher cardinality changes are frequent with 34% of the library pairs requiring at least one higher-cardinality code change [23]. Overall, we find that the LLMs are reasonably successful at migrating them, with 70%, 79%, and 84% correctness rates.

RQ1, Code Change Level: 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.

V. RQ2 How many migrations pass unit tests?

A. Approach

We now assess the same LLM PyMigBench migrations from RQ1 but using the unit tests available in the corresponding repositories. A test-based evaluation that runs the code ensures that the migration achieves the expected behavior. Note that we should not expect that an LLM migration improves the rate of passing tests; a successful migration means that each test that previously passed on code_{dev} must still pass on code_{llm}. We now explain our evaluation set up.

1) Preparing the code: 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.

2) Setting up the virtual environment: 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.

3) Running the tests and coverage: 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. We maintain a configuration file where we record any project-specific configurations or commands we used. We include this information in our artifact.

TABLE IV: Correctness of 25 PyMigBench migrations using their available unit tests (RQ2)

	Number (percentage) of correct 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

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, specially 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$_{dev}$. 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 RQ1.

4) Determining migration status: 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.

B. Findings

Table IV 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%).

RQ2: 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.

VI. RQ3 Can LLMs perform migrations they have not seen before?

A. Approach

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). To validate whether LLMs are able to perform migrations that are not known to them, but for libraries they are aware of, we run an experiment with repositories that never contained the target migration and that are updated after the training cutoff dates. To allow validating the migration, we choose repositories with high test coverage.

We use SEART [39] to find Python repositories that are updated on or after January 1, 2024. We only keep repositories that have a `requirements.txt` file to allow us to set up the environment to run tests. We clone the latest version of the repositories one by one, set up a virtual environment using the requirements file, and then run tests with coverage. We stop after we find 10 repositories whose tests pass and achieve at least 95% statement coverage.

For each repository, we select a library listed in the requirements file and identify an analogous library that provides similar functionality through an online search. Overall, we choose 10 unique library pairs, 4 from PyMigBench and 6 external to it. This allows us to evaluate migration of unseen code, both on library pairs we have seen the models perform on before, as well as on new pairs. To ensure that the models have never seen the expected code that uses the target library in these repositories, we traverse the commit histories of each repository and confirm that the target library never appeared in the requirements file. This also removes the possibility that the LLMs have previously seen an inverse migration from the target library to the source library in the repository's history.

Next, we find the code files that use the source library and thus require migration. For each of the 10 repositories, we parse each file using the Python AST module to find the files that import the source library and thus require migration. We also manually locate the API usages in those files to confirm that it does not just import the library but also uses it. We then run the available tests to record their current status and also manually verify that the API usages of the source libraries are covered by the tests. Finally, we migrate each of the identified files using the same prompt template we used before (Figure 1). We run the tests after the migration and compare the test results to those before the migration.

Overall, for this experiment, we have a total of 10 migrations in 10 repositories between 10 unique library pairs. The repositories have between 1 and 5 migration-related files (median 1) and between 2 to 210 source API usages that require migration (median 6). The number of tests in the 10 repositories ranges from 7 to 903 (median 83), all passing before migration with 95% or more statement coverage.

B. Findings

The migrations have between 3 and 1,125 lines modified per migration (median 36), confirming that the LLMs attempted to migrate the code rather than returning the same code. Figure 3 shows the percentage of passing tests for each migration. The dashed lines show the median passing tests for each model. 4o has the highest median passing tests (100%), followed by Mini (47%). LLama has most of the migrations (70%) fail, resulting in the median passing tests of 0.0%.

4o has 5 fully correct migrations (i.e., all tests passing after migration) plus two more migrations having 98% and 99% tests passing. 4o has only 2 fully incorrect migrations (no tests

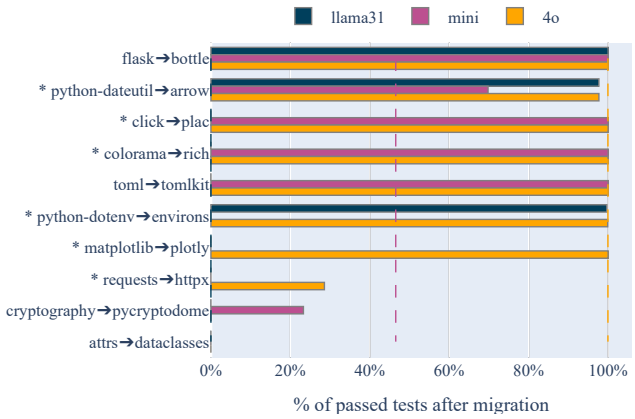


Fig. 3: Correctness of migrations unseen by the LLMs (RQ3). Asterisks (∗) indicate the library pair is not in PyMigBench.

passing after migration). Mini follows 4o with 4 fully correct migrations, and 4 fully incorrect migrations. LLama has only 1 fully correct migration but has two migrations having 98% tests passing. Overall, the relative performance of the three models follow the same trend as RQ1 and RQ2, with 4o being the best, followed by Mini and then LLama.

We now compare RQ3 results with RQ1 for the 4 library pairs that also appear in PyMigBench. This helps us understand whether migrations between these library pairs are inherently challenging or if the LLMs did better on potentially familiar code.

attrs→dataclasses and *cryptography→pycryptodome* migrations illustrate the challenging cases in these four library pairs. We find that these library pairs show consistent poor performance across all RQs. In RQ3, the *attrs→dataclasses* migration results in complete failure across all three models. Similarly in RQ1, only 2 out of 12 attempts for this library pair by the models are fully correct. The *cryptography→pycryptodome* migration completely fails for two models in RQ3, and only 12 out of 45 such instances are correct in RQ1.

In contrast, *toml→tomlkit* and *flask→bottle* represent two consistently easier cases, where the LLMs achieve high or perfect correctness across all RQs. Except for one *flask→bottle* migration where the token limit was exceed for LLama in RQ1, all migrations in these two library pairs are at least partially correct.

Overall, this suggests that the difficulty of a migration is primarily determined by the inherent complexity of the library pairs, not by whether the migration was seen during training. Specifically, library pairs that were hard for the models in RQ3 were also hard in RQ1 and RQ2, while easier pairs consistently achieved better results.

RQ3: Out of 10 unseen migrations, the LLMs fully migrated 1-5 migrations, with 4o being the highest. The performance patterns are consistent with those observed in RQ1.

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

(a) Before migration

```
4 auth = OAuthHandler(consumer_key, consumer_secret)
5 auth.set_access_token(oauth_token, oauth_secret)
6 tw_api = tweepy.API(auth)
```

(b) After LLM migration

Fig. 4: LLama correctly migrating many-to-many code change

VII. Discussion

A. Are LLMs Suitable For Library Migration?

1) The Good: We start with the promising side of using LLMs for library migration. In addition to the high overall correctness of code changes and migrations in RQ1, we find that the LLMs can migrate code changes that were traditionally known as difficult to migrate [40]. We also find that LLMs can perform migrations on code they have not seen before. We discuss some examples to demonstrate these strengths.

a) Different API styles: The library *argparse* [41] provides API functions for parsing command-line arguments, while *click* [36] provides decorators. Despite the completely different API styles, the three LLMs correctly migrate 94% of code changes between this library pair.

b) Higher cardinality migrations: As shown in RQ1, 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 [40, 42]. Figure 4 shows an example of a many-to-many code change in a *twitter→tweepy* migration done by LLama. In addition to correctly replacing 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.

c) Inferring changes outside of the source and target APIs: In an *eventlet→gevent* migration, 4o replaced a call `SocketIO()` with `SocketIO(async_mode='gevent')`. This is interesting because the `SocketIO` API is from the *Flask-SocketIO* library, which is neither the source nor the target. The `SocketIO` function works with several libraries, *eventlet* being the default [43]. If 4o did not set the argument to `gevent`, the code would break after the migration.

d) Unseen migrations: When using LLMs for library migration, we do rely on their knowledge of the APIs. However, we also need to ensure they can apply this knowledge to unseen code and are not just reciting the target code. The results of RQ3 suggest that this is indeed the case, giving us confidence that an LLM-based migration tool can be useful in practice. RQ3 even demonstrates that LLMs can handle large migrations they have not seen before. Specifically, the *attrs→dataclasses* migration in RQ3 is the largest one, with 5 files and 210 source API usages to migrate, where 4o changed 615 lines of code to perform this migration. Figure 3 shows that all three LLMs failed all tests for this migration. However, upon further investigation, we find that 4o was able to migrate most of the code correctly, but the tests failed because the

original code had a default field defined before a non-default field in a class, which is not allowed in the target library. We manually fix this by changing 2 lines of code, resulting in 53% tests passing. This suggests that, in the worst case, LLMs can provide a good starting point even for large (unseen) migrations that may be more challenging.

2) The Bad: We also analyze cases where the LLMs did not correctly migrate the code and present notable observations.

a) Failing to handle argument transformation: The `wait_fixed` argument in `@retry()` from the library *retrying* [44] expects the time in milliseconds, while the target library *tenacity* [45] expects it in seconds. Therefore, it requires transformation, for example, from `@retry(wait_fixed=2000)` to `@retry(wait=wait_fixed(2))`. Out of 28 instances of this change, 4o handled three instances correctly, and the other two models each handled just one instance correctly. This suggests that while the LLMs can map parameters, differences in expected format or units are harder to manage.

b) Replacing exception type: We find cases where the LLM replaces generic exception types with specific ones. e.g., `Exception` with `ValueError`, which can potentially fail to catch the same exceptions as the original code. This occurs in both migration-related and non-migration-related changes. While using more specific exceptions is considered good practice [46], it does change the program behavior. Since modern development tools are often equipped to suggest better exception handling, it may be safer for a migration tool to avoid changing existing exception types.

3) The Ugly: We also observe the following problematic LLM behavior.

a) Extra Changes: LLMs are known for not always following the instructions provided in the prompt [47]. Despite our prompt explicitly asking the LLMs to avoid any non-migration changes, we find cases where an LLM, particularly LLama, completely removes parts of the code that do not include any source APIs. Silently deleting code can be dangerous for the overall functionality of the system, and led to test failures in RQ2 for many cases.

The LLMs also sometimes refactor code that is not related to the migration. While these improvements can be beneficial, they are unrelated to the migration task at hand. Future work could explore the use of an interactive IDE plugin that confirms the code changes before applying them, potentially allowing the developer to directly edit the changes. While less "dangerous", we also find that the LLMs remove code comments and docstrings from the source code, even though the comments remain relevant after migration.

b) Large file handling: We observe that large files pose a challenge to LLama, given its smaller context size. Additionally, we observe that even the other two models often migrate an initial part of a large file but then incorrectly state that the remaining part does not require any changes.

B. Differences in Evaluation Setup

The test-based evaluation in RQ2 uses a subset of RQ1 migrations. Ideally, the same migration in these two RQs should yield the same results. However, in practice, we observe some discrepancies, which we investigate and explain below.

```
4 import ipaddr
5 def IP(ipaddress):
6   a = ipaddr.IPNetwork
    (ipaddress)
```

(a) Before migration

```
4 import ipaddress
5 def IP(ipaddress):
6   a = ipaddress.
    ip_network(ipaddress)
```

(b) After Mini's migration

Fig. 5: Example of parameter shadowing import name.

1) Library version compatibility: In RQ1, when we manually match changes, we mark a code change to be correct based on the documentation of latest version of the library. Since we do not ask for a specific version in the prompt, we notice that the LLMs usually pick APIs from the latest version of the target library, so the LLM's result and manual evaluation align. However, in RQ2, we install the versions applicable at the time of migration (Section V-A). This version discrepency often led to run-time errors when running tests on the LLM's code, resulting in the migration marked as incorrect in RQ2.

2) Function parameter shadowing import name: In Figure 5, Mini correctly replaces the import `ipaddr` with import `ipaddress` and the API `IPNetwork` with `ip_network`; therefore, in RQ1, we count this as a correct migration. However, notice that the parameter in the function `IP` is named `ipaddress`, the same as import name, which causes the parameter to shadow the import name and leads to test failures. The developer and 4o both handled this by changing the parameter name (not shown in the Figure), therefore the tests did not fail for them.

3) Developer doing non-refactoring changes: Recall that in RQ1, we find cases where the developer makes non-refactoring changes in the same commit as the migration while the LLM only does the migration it is expected to do. We count this as a correct migration in RQ1. However, given the changed behavior (often reflected in the tests), the tests fail in RQ2.

C. Understanding the cost of LLM-based migration

The 314 migrations we use have a total of 2,584K lines of code (KLOC) for all 10 runs per models. The cost of running our experiments was approximately $15 for Mini and $420 for 4o, which equates to $0.01 and $0.16 per KLOC, respectively. LLama is free if self-hosted, though we used a pro subscription of *Hugging Face* (US$9/month) to avoid installation overheads. While library migration is an infrequent task in a project's lifetime, it requires significant developer effort when it happens. The cost of a developer's time can easily exceed the cost of using an LLM, even 4o, which can justify the cost of LLM-based tooling.

VIII. Related work

A. Traditional library migration techniques

The majority of automated techniques for library migration focus only on identifying API mapping. Some techniques identify analogous APIs by analyzing *diffs* in migration commits using static analysis [4, 8, 9] and machine learning [42]. Other efforts overcome the need for real migrations by using

unsupervised learning [10] and natural language processing (NLP) of documentation [5]. DeepMig [48] uses a transformer-based architecture to recommend alternative library and a migration plan, but does not perform the transformation.

SOAR [5] applies migration to the client code using program synthesis guided by unit tests. SOAR is evaluated on two pairs of deep learning libraries, focusing on the migration of neural network models. 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. In contrast, due to the diversity of the libraries we evaluate on, we use the unit tests existing in the repositories to evaluate the correctness of the migrated code. SOAR supports one-to-many code changes, but only for a small set of pre-determined APIs. SOAR times out for 20% of the evaluated migrations, even for small code snippets (the largest being 183 lines of code). Our evaluation suggests that LLMs may be able to handle higher-order transformations out of the box for larger code snippets.

B. LLM-based library migration

Almeida et al. [49] evaluate the performance of ChatGPT for *SQAlchemy 1→SQLAlchemy 2* migrations. Nikolov et al. [6] describe Google's experiences in using LLMs for several internal code transformation tasks, including *JUnit 3→JUnit 4* and *Joda Time→Java Time* migrations. They find that using LLMs reduced migration time by at least 50% compared to manual migration. Zhou et al. [7] proposed HaPiM, which first trains a machine learning model capable of API mapping, and then uses the API mappings to guide an LLM in code transformation. Their approach outperforms MigrationMapper [4] and GPT-3.5 Turbo on the BLEU [50] and CodeBLEU [51] metrics evaluated on 5 Java library pairs.

The above studies show that LLMs can be effective for library migration. These studies, however, have several limitations. First, Almeida et al. [49] and Google's *JUnit 3→JUnit 4* migrations [6] explore LLMs capability of migrating from one version of a library to another, while we focus on migrating between different libraries. Migrating versions is different than migrating libraries, because the former can leverage the evolution or release history of the libraries to identify API changes. Second, all three evaluations are based on very limited number of libraries, maximum 5. Google's *Joda Time→Java Time* migrations [6] also provide library specific instructions. Therefore, there is a lack of generalizability. Our evaluation, on the other hand, covers 134 Python library pairs from 34 application domains. HaPiM [7] also requires first to train a machine learning model to identify API mappings using a set of existing migrations, which is a limitation noted in previous studies [5, 10]. Finally, none of the above studies analyze the types of code changes that can be handled.

IX. Threats to Validity

A. Internal validity

Our benchmark-based evaluation in RQ1 relies on PyMigBench labeled data, which may contain errors. The PyMigBench authors, however, manually vetted each code change to ensure correctness. During our manual evaluation, we only found a couple of incorrectly labeled changes, which were eventually corrected in PyMigBench.

Our evaluation involves two manual steps: validating correctness of code changes and labeling a non-migration related change as refactoring or non-refactoring. Manual activities are inherently subjective and may contain errors. We minimize this by having two authors independently perform the manual activities and resolving disagreements through discussion.

B. Construct validity

In RQ1, 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. Categorizing the LLM changes would require additional manual effort following the previous work by Islam et al. [23], which 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.

Tests passing may not always guarantee migration correctness (e.g., the tests do not cover certain edge cases). We mitigate this, to the extent possible, by selecting repositories with high test coverage in RQ3 and ensuring that the tests cover the source API usages.

C. External validity

We use three off the shelf LLMs. While using an LLM specifically tuned for the library migration task could potentially yield better results, the goal of this work is not to develop a migration tool, but to understand the challenges and opportunities of using LLMs for library migration. Our results may be viewed as a lower bound of the performance of potentially more specialized models.

Our results may not generalize beyond Python. We choose Python, because it is a popular language, and because we did not find a dataset that has the same detailed characterization of migration-related code changes for other languages.

PyMigBench may not be representative of all Python libraries. However, the dataset contains a diverse set of 134 library pairs from 34 different application domains. On the other hand, these are all existing libraries where the LLMs are likely familiar with their APIs and API usage. 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 work.

X. Conclusion

This paper presented an empirical study of using LLMs for library migration in Python. Specifically, we use LLama 3.1, GPT-4o mini, and GPT-4o to migrate 314 migration commits across 294 client repositories. We compared the LLM changes to the developer changes recorded in PyMigBench for these migrations. We find that 4o performs the best, and correctly migrates 94% of the individual code changes. At the migration level, it perfectly migrates 57% of the migrations, and 94% of them have at least one correctly migrated code change. We also run unit tests for a subset of the migrations, and find that 64% of the migrations by 4o have the same sets of tests passing in the developer's migration and the LLM's migration. A much lower cost LLM Mini and the free LLama also perform relatively well, with both correctly migrating 89% of code changes. We also find that the LLMs can correctly perform unseen migrations. Overall, our results suggest that using LLMs for library migration is promising, and we discuss the opportunities for further work in this area.

References

[1] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. "Keep me updated: An empirical study of third-party library updatability on android". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2187–2200.

[2] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. "Do developers update their library dependencies?" In: *Empirical Software Engineering* 23.1 (2018), pp. 384–417.

[3] Cedric Teyton, Jean-Remy Falleri, and Xavier Blanc. "Mining library migration graphs". In: *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 289–298.

[4] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. "On the use of information retrieval to automate the detection of third-party java library migration at the method level". In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 347–357.

[5] Ansong Ni et al. "Soar: a synthesis approach for data science api refactoring". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 112–124.

[6] Stoyan Nikolov et al. "How is Google using AI for internal code migrations?" In: *arXiv preprint arXiv:2501.06972* (2025).

[7] Bingzhe Zhou et al. "Hybrid API migration: A marriage of small API mapping models and large language models". In: *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 2023, pp. 12–21.

[8] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. "Automatic discovery of function mappings between similar libraries". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE. 2013, pp. 192–201.

[9] Hussein Alrubaye and Mohamed Wiem Mkaouer. "Automating the detection of third-party Java library migration at the function level." In: *CASCON*. 2018, pp. 60–71.

[10] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. "Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding". In: *IEEE Transactions on Software Engineering* 47.3 (2019), pp. 432–447.

[11] Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. "Deep-diving into documentation to develop improved java-to-swift api mapping". In: *Proceedings of the 28th International Conference on Program Comprehension*. 2020, pp. 106–116.

[12] Ipek Ozkaya. "Application of large language models to software engineering tasks: Opportunities, risks, and implications". In: *IEEE Software* 40.3 (2023), pp. 4–8.

[13] Angela Fan et al. "Large language models for software engineering: Survey and open problems". In: *arXiv preprint arXiv:2310.03533* (2023).

[14] Junjie Wang et al. "Software testing with large language models: Survey, landscape, and vision". In: *IEEE Transactions on Software Engineering* (2024).

[15] Nhan Nguyen and Sarah Nadi. "An empirical evaluation of GitHub copilot's code suggestions". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 1–5.

[16] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. "The impact of ai on developer productivity: Evidence from github copilot". In: *arXiv preprint arXiv:2302.06590* (2023).

[17] Vijayaraghavan Murali et al. "CodeCompose: A large-scale industrial deployment of AI-assisted code authoring". In: *arXiv preprint arXiv:2305.12050* (2023).

[18] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. "Using an LLM to Help With Code Understanding". In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society. 2024, pp. 881–881.

[19] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. "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*. 2023, pp. 172–184.

[20] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. "Automated program repair in the era of large pre-trained language models". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 1482–1494.

[21] Bingzhe Zhou et al. “Hybrid API migration: A marriage of small API mapping models and large language models”. In: *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 2023, pp. 12–21.

[22] Mohayeminul Islam, Ajay Kumar Jha, Sarah Nadi, and Ildar Akhmetov. “PyMigBench: A Benchmark for Python Library Migration”. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE. 2023, pp. 511–515. DOI: 10.1109/MSR59073.2023.00075.

[23] Mohayeminul Islam, Ajay Kumar Jha, Ildar Akhmetov, and Sarah Nadi. “Characterizing Python Library Migrations”. In: *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*. 2024. DOI: 10.1145/3643731.

[24] Abhimanyu Dubey et al. “The llama 3 herd of models”. In: *arXiv preprint arXiv:2407.21783* (2024).

[25] OpenAI. https://platform.openai.com/docs/models/gpt-4o-mini.

[26] OpenAI. *OpenAI models: GPT-4o*. https://platform.openai.com/docs/models/gpt-4o. 2025.

[27] Meta. *Meta Llama*. https://llama.meta.com/. 2025.

[28] OpenAI. *OpenAI models*. https://platform.openai.com/docs/models. 2024.

[29] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. “PyDriller: Python framework for mining software repositories”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. ISBN: 9781450355735. DOI: 10.1145/3236024.3264598. URL: http://dl.acm.org/citation.cfm?doid=3236024.3264598.

[30] Python Software Foundation. *ast — Abstract Syntax Trees*. https://docs.python.org/3/library/ast.html. 2024.

[31] Berk Atil et al. *LLM Stability: A detailed analysis with some surprises*. 2024. arXiv: 2408.04667 [cs.CL]. URL: https://arxiv.org/abs/2408.04667.

[32] The Git Development Community. *git-diff*. https://git-scm.com/docs/git-diff. 2024.

[33] Chunyang Chen. “Similarapi: mining analogical apis for library migration”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 2020, pp. 37–40.

[34] Jacob Cohen. “A coefficient of agreement for nominal scales”. In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46.

[35] J Richard Landis and Gary G Koch. “The measurement of observer agreement for categorical data”. In: *biometrics* (1977), pp. 159–174.

[36] Pallets. *click*. https://pypi.org/project/click. 2024.

[37] Chenglong Wang et al. “Transforming Programs between APIs with Many-to-Many Mappings”. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 25:1–25:26. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.25. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6119.

[38] Zhenfei Huang et al. “Mapping APIs in Dynamic-typed Programs by Leveraging Transfer Learning”. In: *ACM Trans. Softw. Eng. Methodol.* (Jan. 2024). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3641848. URL: https://doi.org/10.1145/3641848.

[39] O. Dabic, E. Aghajani, and G. Bavota. “Sampling Projects in GitHub for MSR Studies”. In: *2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 560–564. DOI: 10.1109/MSR52588.2021.00074. URL: https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00074.

[40] Shengzhe Xu, Ziqi Dong, and Na Meng. “Meditor: inference and application of API migration edits”. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 335–346.

[41] Python Software Foundation. *argparse*. https://docs.python.org/3/library/argparse.html. 2024.

[42] Hussein Alrubaye et al. “Learning to recommend third-party library migration opportunities at the API level”. In: *Applied Soft Computing* 90 (2020), p. 106140.

[43] Flask. https://flask-socketio.readthedocs.io/en/latest/api.html. 2024.

[44] Rory Geoghegan. *retrying*. https://pypi.org/project/retrying. 2024.

[45] Johann Schmitz. *tenacity*. https://pypi.org/project/tenacity. 2024.

[46] Nélio Cacho et al. “How does exception handling behavior evolve? an exploratory study in java and c# applications”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 31–40.

[47] Hongbin Ye, Tong Liu, Aijia Zhang, Wei Hua, and Weiqiang Jia. “Cognitive mirage: A review of hallucinations in large language models”. In: *arXiv preprint arXiv:2309.06794* (2023).

[48] Juri Di Rocco et al. “DeepMig: A transformer-based approach to support coupled library and code migrations”. In: *Information and Software Technology* 177 (2025), p. 107588.

[49] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. “Automatic Library Migration Using Large Language Models: First Results”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2024, pp. 427–433.

[50] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th*

annual meeting of the Association for Computational Linguistics. 2002, pp. 311–318.

[51] Shuo Ren et al. “Codebleu: a method for automatic evaluation of code synthesis”. In: *arXiv preprint arXiv:2009.10297* (2020).