

# Pig: Leveraging Large Language Models for Python Library Migrations

MIRYEONG KANG, Korea University, Republic of Korea

WONSEOK OH, Korea University, Republic of Korea

GABIN AN, Korea University, Republic of Korea

HAKJOO OH, Korea University, Republic of Korea

We present Pig, a novel approach to automating Python library migration by leveraging large language models (LLMs). Library migration is an increasingly common task in modern Python development, yet it remains tedious and error-prone due to the lack of general solutions that can handle diverse libraries without relying on documentation or code examples. To address this challenge, Pig employs a four-step pipeline that effectively harnesses the capabilities of LLMs. First, Pig decomposes the migration task into smaller units by performing API-level slicing, allowing the LLM to focus on minimal, relevant context. Second, it guides LLMs using prompts informed by common failure patterns in naive LLM-based migrations and plausible API candidates. Third, Pig selectively extracts the migration-related code fragments from the LLM outputs. Finally, it transplants the migrated code back into the original program with post-processing to ensure semantic correctness and consistency. We demonstrate the effectiveness of Pig by evaluating it on 364 API-level migration tasks, where it improves the average success rate of the baseline approach by 53.5% across seven different LLM models.

CCS Concepts: • **Software and its engineering** → **Maintaining software**.

Additional Key Words and Phrases: Library Migration, Python, Large Language Model, Code Transformation

## ACM Reference Format:

Miryeong Kang, Wonseok Oh, Gabin An, and Hakjoo Oh. 2026. Pig: Leveraging Large Language Models for Python Library Migrations. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE044 (July 2026), 22 pages. <https://doi.org/10.1145/3797072>

## 1 Introduction

Application Programming Interfaces (APIs) in third-party libraries help developers by "encapsulating" complex domain-specific functionality. During development, developers often need to replace APIs from one library with those from a different library or version (i.e., Library Migration). This need can arise due to discontinued maintenance, the availability of superior alternatives, compatibility issues, or shifts in community support. As such, library migration is a common and essential task in software development for adopting more effective libraries and maintaining software quality.

However, substituting the original APIs while preserving the intended behavior of the program is a non-trivial task: it requires deep understanding of both the source and target APIs, as well as the ability to produce semantically equivalent code. Moreover, migration often entails modifying not just API function calls but also surrounding code, adjusting data transformations, control flow, or usage patterns, rather than simply replacing identifiers. These challenges make library migration

---

Authors' Contact Information: [Miryeong Kang](mailto:miryeong59@korea.ac.kr), Korea University, Seoul, Republic of Korea, [miryeong59@korea.ac.kr](mailto:miryeong59@korea.ac.kr); [Wonseok Oh](mailto:wonseok_oh@korea.ac.kr), Korea University, Seoul, Republic of Korea, [wonseok\\_oh@korea.ac.kr](mailto:wonseok_oh@korea.ac.kr); [Gabin An](mailto:gabin_an@korea.ac.kr), Korea University, Seoul, Republic of Korea, [gabin\\_an@korea.ac.kr](mailto:gabin_an@korea.ac.kr); [Hakjoo Oh](mailto:hakjoo_oh@korea.ac.kr), Korea University, Seoul, Republic of Korea, [hakjoo\\_oh@korea.ac.kr](mailto:hakjoo_oh@korea.ac.kr).



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

© 2026 Copyright held by the owner/author(s).

ACM 2994-970X/2026/7-ARTFSE044

<https://doi.org/10.1145/3797072>

a time-consuming and error-prone task for developers [26]. Thus, automating this process can help reduce the manual effort involved and minimize the risk of introducing bugs. In Python, the need for migration is further amplified by the rapid pace of library evolution across domains and the ecosystem's rich and diverse library landscape [19].

**Existing Work.** Existing migration techniques for Python, such as the one proposed by Ni et al. [32], suffer from limited generality across libraries. Their method targets migration between machine learning libraries by leveraging library documentation and error messages. However, the technique assumes the consistent availability of such artifacts, which cannot be guaranteed for all libraries. Furthermore, the use of error messages is domain-specific, further limiting the technique's applicability. Indeed, Gu et al. [19] found that most library migrations in Python go beyond the machine learning domain and include a wide range of general-purpose libraries. Research on library migration in Java [14, 27, 45], which is not tied to a specific domain, typically requires large sets of pre-existing code examples, which imposes a significant burden on developers to collect or curate such data. In particular, the practical applicability of such approaches is limited in Python, where collecting a sufficiently large and diverse corpus of migration examples is often intractable [22].

To overcome the limitations of generality and applicability in existing works, recent studies have explored using Large Language Models (LLMs) for library migration [1, 23]. Almeida et al. [1] found that LLM library migration performance can be substantially improved by using a one-shot setting (i.e., providing an example of migrated code in the prompt) or Chain-of-Thought (CoT) prompting (i.e., providing explicit instructions on how to perform the migration). While these strategies enhance migration accuracy, they are often impractical in real-world scenarios, because curating relevant examples or crafting migration-specific instructions for each case imposes significant manual effort, thereby undermining the goal of automation. In contrast, Islam et al. [23] explored a zero-shot setting, prompting LLMs to migrate entire files without any auxiliary examples or instructions. Their evaluation highlighted the promise of this approach, but also exposed its limitations, such as unintended semantic changes and omissions of essential code segments. In this work, we seek to strike a balance between these two ends of the spectrum. We explore how to effectively leverage LLMs for library migration in a practical zero-shot setting, i.e., avoiding reliance on handcrafted examples or instructions, while improving the reliability and quality of the migration outputs.

**Our Approach.** In this paper, we present PiG (Python Library Migration), a novel approach for automating library migration in Python using LLMs, without relying on any guidance or pre-provided code examples. PiG aims to maximize the effectiveness of LLMs in the migration process: it uses LLMs to generate draft migration code, followed by a series of automated post-processing (or automated fixes) that enhance the reliability and accuracy of the final result. The approach consists of four main stages: First, PiG avoids delegating the entire migration task to the LLM. Instead, it decomposes the original migration task into smaller units by requesting migrations on a per-API basis, each scoped to minimized context. Second, PiG queries the LLM with a tailored prompt containing plausible API candidates and specific instructions, informed by an analysis of common error patterns in naive LLM outputs. Third, PiG identifies and extracts the migration-relevant lines from LLM-generated code, ensuring the semantic correctness of the surrounding context. Finally, PiG transplants the extracted migration code into the original source, applying additional post-processing to ensure the correctness of the final code.

**Results.** We implement PiG and evaluate it on 69 real-world, file-level library migration tasks from PyMIGBENCH2.0 [21], comprising 364 API-level migration tasks in total. Compared to the recent LLM-based approach [23], PiG achieves a 53.5% higher average success rate at the API level

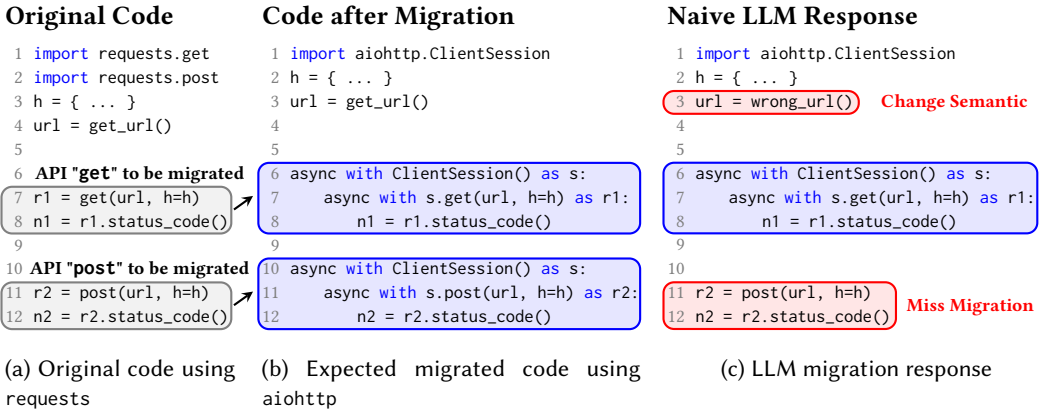


Fig. 1. An example of the migration code from requests to aiohttp. The LLM response for the migration request fails to migrate the second API call and introduces wrong semantic at line 3.

across seven different LLM models. The evaluation further shows that all stages in Pig contribute positively to its overall performance. We also evaluate Pig on a dataset carefully curated to avoid data leakage and find that it achieves notable improvements on unseen code, demonstrating its generalizability and robustness. Finally, we conduct a qualitative analysis on Pig’s failure cases, highlighting key challenges and future directions for enhancing LLM-based library migration.

**Contributions.** Our contributions are as follows:

- We present Pig, a novel method for automating Python library migration using only LLMs, powered by subtask decomposition, prompt optimization, and context-aware post-processing, without relying on external documentation or code examples.
- We evaluate Pig on real-world library migration tasks and show that Pig outperforms the naive approach of using LLM for the whole migration.
- We construct a benchmark for library migration that is free from data leakage, enabling evaluation on unseen code and providing a more realistic assessment.
- We investigate the primary causes of migration failures in LLMs and outline current challenges along with future directions for improving LLM-based library migration.

## 2 Overview

This paper addresses the migration of dependencies within a single Python file, focusing on replacing multiple API calls from one library with equivalents from another. Recent studies have shown that LLM performance tends to degrade as both the number of APIs to be migrated and the code size increase [23].

Figure 1 illustrates a motivating example in which we aim to migrate the get and post APIs from the requests library (Figure 1a) to their counterparts in aiohttp (Figure 1b). As shown in Figure 1c, naively prompting an LLM with the original code often yields incorrect results; for instance, changing semantic or omitting necessary API changes. This highlights a key technical challenge: LLMs often struggle to accurately perform multiple simultaneous API transformations.

To better understand the limitations of LLMs in multi-API migration, we conducted an empirical study involving seven different open-source LLMs. Using prompt templates adapted from prior work [23], we executed 69 unique multi-API migration tasks drawn from the PyMigBench2.0 [21], which encompasses a total of 364 individual API migration instances. Among these, we identified 76

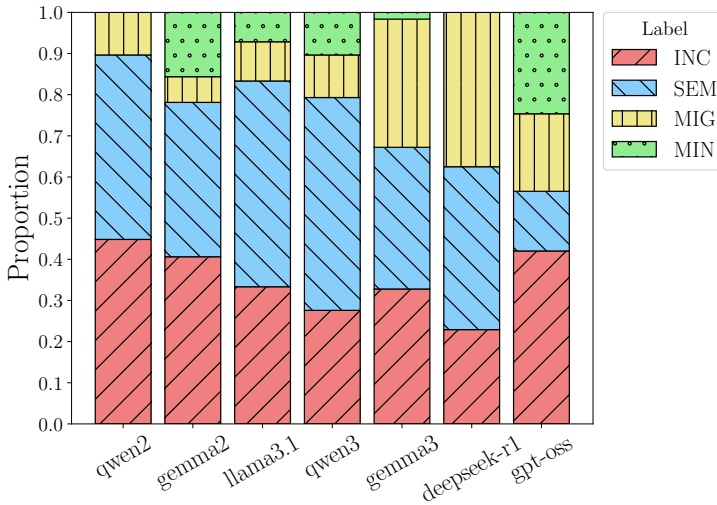


Fig. 2. Error type proportions in migration outputs per model.

instances where all seven models consistently failed to produce correct migration outputs, indicating particularly challenging or nontrivial transformation cases. We focused our error analysis on the outputs generated for these common 76 instances in 7 models, yielding a total of  $76 \times 7 = 532$  migration attempts.

Of the 532 outputs, 281 contained plausible code snippets, i.e., they went beyond mere textual suggestions or high-level commentary and resembled executable code. We conducted a manual, open-coded analysis of these 281 outputs to classify the types of errors made by the models. As shown in Figure 2, we categorized the errors into four primary types, allowing each instance to be associated with multiple error categories as appropriate.

- **INC** (116): Incorrect API usage, where the model generates calls to incorrect APIs or provides invalid arguments.
- **SEM** (126): Semantic changes, where the model alters the semantics of the original code.
- **MIG** (65): Migration omissions, where the model uses a different library or omits the migration of the target API.
- **MIN** (32): Minor errors, where the model introduces small syntax issues or undeclared variables.

Incorrect API usage (**INC**) and semantic changes (**SEM**) were the most prevalent error types, suggesting that LLMs frequently struggle with both API-level precision and the preservation of original semantic. Migration omissions (**MIG**) further underscore the difficulty of achieving full coverage in multi-API transformations, while minor issues (**MIN**), though less frequent, still hinder successful migration and reveal fragility in generated code.

To address these limitations, we propose **PIG**, a four-stage workflow for automated library migration. **PIG** is designed to reduce error rates by narrowing the migration context, guiding model behavior, and carefully transplanting migrated APIs. Specifically, it first applies slicing to help the LLM focus on the relevant migration context (**Stage 1**), instructs the LLM to avoid the common error types and provides plausible API candidates to address **INC** errors (**Stage 2**), and transplants only the migrated API while preserving the semantics to prevent **SEM** errors (**Stage 3-4**). We illustrate the complete **PIG** workflow with an example in this section.

**Sub-Task 1**

```

1 import requests.get
2 h = { ... }
3 url = get_url()
4
5
6 r1 = get(url, h=h)
7 n1 = r1.status_code()

```

**Sub-Task 2**

```

1 import requests.post
2 h = { ... }
3 url = get_url()
4
5
6 r2 = post(url, h=h)
7 n2 = r2.status_code()

```

(a) Sliced code for API get (**Sub-Task 1**) and post (**Sub-Task 2**).

**LLM Response for Sub-Task 1**

```

1 from aiohttp.wrong_path import ClientSession
2 headers = { ... }
3 url = get_url()
4
5 async with ClientSession() as s:
6     async with s.get(url, h=headers) as r1:
7         n1 = r1.status_code()

```

**LLM Response for Sub-Task 2**

```

1 from aiohttp.wrong_path import ClientSession
2 h = { ... }
3 url = wrong_url()
4
5 async with ClientSession() as s:
6     async with s.post(url, h=h) as r2:
7         n2 = r2.status_code()

```

(b) LLM responses for the two sub-tasks shown in Figure 3. Migrated code and incorrect code are highlighted in blue and red, respectively.

Fig. 3. The decomposition of the original source code into API-level sub-tasks via slicing.

**Stage 1: Decomposing Migration Tasks with Slicing.** Prior work [21] reports that migrating a single Python file typically involves around seven API updates. Given this complexity, our preliminary experiments show that supplying the entire file to an LLM results in low migration success. Motivated by this, we adopted an API-level decomposition strategy: dividing the file into smaller, API-level sub-tasks leads to a 35.9% improvement in migration success over the naive approach. For instance, as shown in Figure 3a, we divide the original code into two sub-tasks, each centered on a single API call. This fine-grained slicing allows the LLM to operate on a smaller, more specific context, thereby improving the quality and accuracy of the generated migrations. Each sub-task is treated as an independent migration unit and passed through the subsequent stages of our pipeline.

**Stage 2: Querying LLM with plausible API Candidates.** We embed explicit instructions in the LLM prompt to enforce semantic preservation and ensure completeness, aiming to prevent SEM, MIG, and MIN errors. To address INC errors, the most critical, we provide a curated list of plausible API candidates to guide the model toward valid replacements. Notably, source and target library APIs often share syntactic features such as common prefixes, suffixes, or structural elements (e.g., LevelDB vs. DB, or IPAddress vs. ip\_address). This trend appears to hold broadly across libraries. For example, in our benchmark, among 257 source-target API pairs, excluding the case where APIs are deleted, 196 pairs showed syntactic similarity between the source and target API names.

Building on this insight, we developed a candidate identification method based on both name and argument similarity. We first compute the lexical similarity between the source API name and each candidate, resolving ties using argument-level similarity. This method proved effective: in 151 out of 257 cases, the correct target API appeared in the top-3, and in 124 cases, it ranked first. For instance, when migrating `requests.get`, our method ranks `aiohttp.ClientSession.get` as the top candidate, which is the correct replacement.

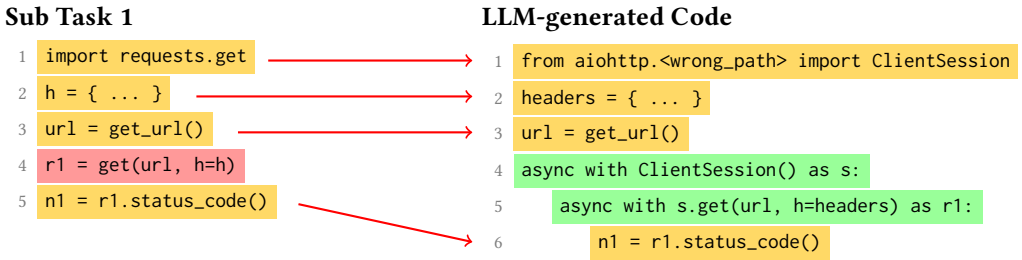


Fig. 4. Example of the standard tree difference algorithm (*GumTree*). The left side shows the original code, while the right side displays the LLM-generated code with modifications.

Therefore, we incorporate the top-3 candidates into the prompt as contextual cues to guide the LLM toward accurate API selection. Upon querying the LLM with our constructed prompt, we receive the output code shown in Figure 3b.

**Stage 3: Identifying the Migrated API Statements.** Given the LLM-generated code snippets for each sub-task (e.g., Figure 3b), we must integrate each migrated API back into a single, unified file. However, this is non-trivial due to SEM errors: the LLM alters surrounding context, making simple *copy-and-paste* approach insufficient.

Figure 3b illustrates why copy-and-paste is inadequate. Although the API migrations succeed in both sub-tasks (lines 5–7), inconsistencies in the surrounding context prevent direct transplantation. In Sub-Task 1 (line 2), the original variable `h` is renamed to `headers`; in Sub-Task 2 (line 3), the value of `url` is altered. These changes risk introducing unintended behavior or breaking program logic. To mitigate this, we transplant only the migration-relevant fragments (specifically, lines 5–7 in Figure 3b) rather than the entire LLM-generated block. We use a tree differencing algorithm to detect structural changes and accurately isolate the migrated API statements.

However, standard tree differencing also struggles with the variability in API usage patterns. For instance, Figure 4 presents the differencing results between the original and LLM-generated code, where yellow, red, and green highlight modified, removed, and added code, respectively. The original call `r = get(url, headers=h)` (line 4, left) is transformed into `async with s.get(url, headers=headers) as r1:` (line 5, right). Due to this structural shift, the algorithm misclassifies the original API as removed. This example highlights a key limitation of standard tree differencing: difficulty in aligning structurally divergent but semantically equivalent code.

In response, we focus on a characteristic of API migration to bridge this structural gap. Islam et al. [21] reveals that 88.17% of migration instances retain their syntactic structure. Particularly, for migrations between function calls, 82.16% of cases preserve the original arguments without any deletions. Based on this insight, we identify the migrated API statement as the one with the *highest sub-expression similarity*, rather than relying solely on statement-level matches. For example, in Figure 4, `get(...)` becomes `s.get(...)`, retaining the call structure, while the argument `url` and the keyword argument `h` also persist in the migrated code. This allows us to correctly identify line 5 as the migration-relevant statement.

**Stage 4: Transplanting the Migrated Code.** The migration-relevant statements identified in the previous stage must be transplanted into the original source code. However, this process may introduce undefined variables that did not exist in the original context. For instance, in Figure 4, the migrated statement `async with s.get(path, headers=headers) as r1:` uses new variables

```

1 # The import statement for the migrated API
2 from aiohttp import ClientSession
3
4 # The unchanged original code
5 h = { ... }
6 url = get_url()
7
8 # Sub-task 1
9 async with ClientSession() as s:
10     async with s.get(url, h=h) as r1:
11         n1 = r1.status_code()
12
13 # Sub-task 2
14 async with ClientSession() as s:
15     async with s.post(url, h=h) as r2:
16         n2 = r2.status_code()

```

Fig. 5. Final code after migration. Transplanted and unchanged original code are highlighted in blue and gray, respectively.

`s` and `headers`. We address this issue using two strategies: (1) variable renaming and (2) context augmentation.

**(1) Variable Renaming.** LLMs sometimes introduce renamed variables that correspond to existing ones in the original code. For instance, `headers` in the LLM output aligns with `h` in the original code. To preserve consistency, we revert such renamings. First, we locate the declaration of the new variable in the LLM-generated code (e.g., `headers = { ... }`) and use the tree differencing algorithm introduced in Stage 3 to find its counterpart in the original code (e.g., `h = { ... }`). Based on this alignment, we rename `headers` back to `h`, preserving the original semantics and preventing errors from unintended identifier changes.

**(2) Context Augmentation.** Unlike `headers`, the variable `s` is entirely new and has no corresponding declaration in the original code. To resolve this, we transplant its declaration, e.g., `async with ClientSession() as s:`, into the migrated code. We first attempt to match this statement against the original source, but the tree differencing algorithm classifies it as newly introduced rather than transformed. As a result, we treat it as a context statement that must be added to avoid undefined variable errors.

To ensure completeness, we recursively examine each added context statement for further undefined variables. In this example, `ClientSession`, a class from the `aiohttp` library, is not defined in the original code, prompting an additional context augmentation step: including the appropriate import statement. However, even when the LLM generates a necessary import as part of its output, it often misidentifies module paths, resulting in incorrect imports. For example, it may produce `aiohttp.<wrong_path>`, which does not correspond to any valid module within the `aiohttp` library.

To address this, we first validate the generated import path against the target library's implementation. If the path is invalid, we enumerate all valid module paths through which the target API can be accessed and compute string similarity scores to identify the closest match. For example, in the case of `ClientSession`, since the path `aiohttp.<wrong_path>` does not exist in the `aiohttp` library, we retrieve all valid import paths through which `ClientSession` can be accessed and, based on similarity scoring, identify `from aiohttp import ClientSession` as the correct path.

**Algorithm 1** Overall Algorithm of PiG**Input:** A migration task  $(P_{old}, \mathcal{A}_{old}, L_{old}, L_{new})$ **Output:** A migrated program  $P_{new}$ 

- 1:  $\mathcal{M} \leftarrow \emptyset$
- 2:  $T \leftarrow \text{SubTasks}(P_{old}, \mathcal{A}_{old})$  ▶ Section 3.1
- 3: **for**  $(P_\alpha, \alpha) \in T$  **do**
- 4:    $P_\alpha^{llm} \leftarrow \text{Query}(P_\alpha, \alpha, L_{old}, L_{new})$  ▶ Section 3.2
- 5:    $\mathcal{M}' \leftarrow \text{Actions}(P_\alpha, P_\alpha^{llm}, \alpha, L_{new})$  ▶ Section 3.3
- 6:    $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}'$
- 7:  $P_{new} \leftarrow \mathcal{M}(P_{old})$  ▶ Applying migration actions

**Final Output.** Through these stages, we successfully transplant the LLM-generated migration into the original Python code while preserving its semantics, as illustrated in Figure 5. From the original LLM outputs for each subtask, only the migrated API statements are transplanted into their corresponding locations in the source code, while necessary post-processing steps, such as variable renaming and context augmentation, are applied to ensure correctness.

### 3 The PiG Algorithm

In this section, we describe our approach in detail.

**Problem Definition.** We define an API  $\sigma \in \text{Api}$  as a tuple  $(\alpha, \text{path}, \vec{p})$ , where  $\alpha \in \text{ApiName}$  is the API name,  $\text{path} \in \text{Path}$  is the module path containing the API, and  $\vec{p} \subseteq \text{Var}$  is the set of parameter names of the API. A library  $L \subseteq \text{Api}$  is defined as a set of such APIs.

We define a migration task as a tuple  $(P_{old}, \mathcal{A}_{old}, L_{old}, L_{new})$ , where  $P_{old}$  is the original program,  $\mathcal{A}_{old} \subseteq \text{ApiName}$  is the set of API names used in  $P_{old}$  that are to be migrated (i.e.,  $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$ ),  $L_{old}$  is the source library, and  $L_{new}$  is the target library.

Given a migration task  $(P_{old}, \mathcal{A}_{old}, L_{old}, L_{new})$ , our goal is to generate a new program  $P_{new}$  that uses APIs from the target library  $L_{new}$  in place of those from the source library  $L_{old}$ . To this end, we compute a set of *migration actions*  $\mathcal{M} \subseteq \text{MigrateAction}$  that, when applied to the original program  $P_{old}$ , produces the migrated program  $P_{new}$ . We consider the following types of migration actions:

- $\text{Change}(x, y)$  replaces the variable  $x$  by the variable  $y$ .
- $\text{Modify}(S, S')$  replaces the statement  $S$  with  $S'$ .
- $\text{Delete}(S)$  removes the statement  $S$  from the program.

**Overall Algorithm.** Algorithm 1 presents the overall algorithm of PiG, which consists of the following steps: (1) **Sub-Task Decomposition** (SubTasks, line 2): We decompose the given migration task  $(P_{old}, \mathcal{A}_{old}, L_{old}, L_{new})$  into a set  $T$  of smaller sub-tasks  $(P_\alpha, \alpha)$ . (2) **LLM Query** (Query, line 4): For each sub-task  $(P_\alpha, \alpha)$ , we use an LLM to generate the migrated code  $P_\alpha^{llm}$ . (3) **Action Generation** (Actions, line 5): We generate a set  $\mathcal{M}$  of migration actions that transplant each LLM-generated code back into the original program. This step corresponds to Stages 3 and 4 in Section 2. At line 7, the final program  $P_{new}$  is obtained by applying all migration actions collected in  $\mathcal{M}$  to  $P_{old}$ . The following subsections describe each step in detail.

#### 3.1 Sub-Task Decomposition

A sub-task is a tuple  $(P_\alpha, \alpha)$ , where  $\alpha$  is the API name to be migrated and  $P_\alpha$  is the code segment relevant to the API  $\alpha$ . Given a program  $P_{old}$  and a set  $\mathcal{A}_{old}$  of APIs to be migrated, we decompose

the migration task into API-level sub-tasks:

$$\text{SubTasks}(P_{\text{old}}, \mathcal{A}_{\text{old}}) = \bigcup_{\alpha \in \mathcal{A}_{\text{old}}} \{(P_{\alpha}, \alpha) \mid P_{\alpha} = \text{Slice}(P_{\text{old}}, \alpha)\}$$

where  $\text{Slice}(P_{\text{old}}, \alpha)$  extracts the code segment from  $P_{\text{old}}$  that is relevant to the API  $\alpha$  by performing backward and forward code slicing: (1) **Backward Slicing** starts from the API call site and traces backward to identify all statements that influence the API's usage, and (2) **Forward Slicing** collects statements that directly depend on the output of the API call. Both slices are computed based on def-use chains.

*Example 3.1.* Consider the following program  $P$  where `libo.api` is the API  $\alpha$  to be migrated:

1 <code>import libo</code>	<b>Backward Slicing:</b>
2 <code>a = 1</code>	<code>import libo</code>
3 <code>b = 2</code>	<code>a = 1</code>
4 <code>c = a + 3</code>	<code>c = a + 3</code>
5	<b>Target API:</b>
6 <code>o = libo.api(a)</code>	<code>o = libo.api(a)</code>
7 <code>x = o + 1</code>	<b>Forward Slicing:</b>
8 <code>y = x + 2</code>	<code>x = o + 1</code>

In this example, backward slicing captures all lines related to the API usage (lines 1, 2, and 4), while forward slicing identifies the statement that directly uses the API output (line 7). Thus,  $\text{Slice}(P, \alpha)$  includes lines 1, 2, 4, 6, and 7.

### 3.2 LLM Query

Given a sub-task  $(P_{\alpha}, \alpha)$ , we invoke an LLM to generate the migrated code  $P_{\alpha}^{\text{llm}}$  for the sub-task using the function  $\text{Query}(P_{\alpha}, \alpha, L_{\text{old}}, L_{\text{new}})$ . This function queries the LLM with the code segment  $P_{\alpha}$ , the API  $\alpha$  to be migrated, and the original and target libraries  $L_{\text{old}}$  and  $L_{\text{new}}$ . To this end, we generate the following prompt for the LLM based on the error types we observed (Section 2):

#### Query for sub-task

**(MIG)** You will be tasked with reimplementing a Python code snippet, migrating it from  $\langle L_{\text{old}} \text{ name} \rangle$  to  $\langle L_{\text{new}} \text{ name} \rangle$ . **(SEM)** Do not omit any migration lines, even if some lines are deleted during the rewrite. **(MIN)** Ensure the final code is syntactically correct and free of errors. Migrate the following Python code snippet from the  $\langle L_{\text{old}} \text{ name} \rangle$  to the  $\langle L_{\text{new}} \text{ name} \rangle$ , replacing the API  $\alpha$ . Here is the code snippet to be migrated:  $P_{\alpha}$

**Extracting Plausible API Candidates.** To improve the effectiveness, we provide the LLM with the top-3 plausible API candidates  $\mathcal{A}_{\text{cand}}$  from the target library  $L_{\text{new}}$ . We observed that the API names and parameter names are often similar before and after migration. Based on this insight, we use string similarity measures to identify  $\mathcal{A}_{\text{cand}}$ , a set of likely replacement APIs in  $L_{\text{new}}$ .

To achieve this, we first construct a set  $\phi \subseteq \text{Api} \times (\mathbb{R} \times \mathbb{R})$ , where each element is a tuple consisting of an API from the target library  $L_{\text{new}}$  and a tuple of its corresponding plausibility scores for the name and arguments:

$$\phi = \{(\sigma_{\text{new}}, \text{Sim}(\sigma_{\text{old}}, \sigma_{\text{new}})) \mid \sigma_{\text{new}} \in L_{\text{new}}, \sigma_{\text{new}} = (\alpha_{\text{new}}, \_ , \_)\}$$

where  $\sigma_{\text{old}}$  denotes the API with name  $\alpha$  to be migrated in the current sub-task (i.e.,  $\sigma_{\text{old}} = (\alpha, \_ , \_)$ ) and  $\text{Sim}(\sigma_{\text{old}}, \sigma_{\text{new}})$  measures the similarity between the old and new APIs:

$$\text{Sim}((\alpha_{\text{old}}, \_ , \vec{p}_{\text{old}}), (\alpha_{\text{new}}, \_ , \vec{p}_{\text{new}})) = (\text{StrSim}(\alpha_{\text{old}}, \alpha_{\text{new}}), \text{ParamSim}(\vec{p}_{\text{old}}, \vec{p}_{\text{new}}))$$

where  $StrSim$  calculates a string similarity score based on the Ratcliff/Obershelp algorithm [6].  $ParamSim(\vec{p}_{old}, \vec{p}_{new})$  computes the similarity between parameters  $\vec{p}_{old}$  and  $\vec{p}_{new}$  by applying the Hungarian algorithm [25] to find the optimal one-to-one matching between parameters that maximizes the total sum of the string similarity  $StrSim$  between each matched pair.

*Example 3.2.* Given two parameter sets,  $\vec{p}_{old} = \{u\_name, u\_age\}$  and  $\vec{p}_{new} = \{name, age, phone\}$ . As shown in the following table, the Hungarian algorithm identifies **Case 1** as the optimal mapping with a maximum total similarity score of **1.55**.

Case	Mapping Pairs $(a_i, b_j)$	Total Score
Case 1	$\{(u\_name, name), (u\_age, age)\}$	1.55
Case 2	$\{(u\_name, name), (u\_age, phone)\}$	1.00
Case 3	$\{(u\_name, age), (u\_age, name)\}$	0.88
Case 4	$\{(u\_name, age), (u\_age, phone)\}$	0.64
Case 5	$\{(u\_name, phone), (u\_age, name)\}$	0.80
Case 6	$\{(u\_name, phone), (u\_age, age)\}$	1.11

Additionally, we penalize large differences in the number of parameters. As a result, the function  $ParamSim$  is defined as

$$ParamSim(\vec{p}_{old}, \vec{p}_{new}) = \frac{\text{Hungarian}(\vec{p}_{old}, \vec{p}_{new})}{\sqrt{||\vec{p}_{old}| - |\vec{p}_{new}|| + 1}}$$

where  $\text{Hungarian}(\vec{p}, \vec{p}_{new})$  computes the total sum of the string similarity scores for the optimal matching between parameters.

Next, we select the top-3 candidates based on the scores in  $\phi$ :  $\mathcal{A}_{cand} = \{(\sigma, s) \in \phi \mid s \in Top_3(\phi)\}$  where  $Top_3(\phi)$  denotes the three highest scores among all  $(\sigma, s) \in \phi$  according to lexicographic comparison. We then include the plausible API candidates  $\mathcal{A}_{cand}$  in the prompt provided to the LLM:

#### Prompt for plausible API candidates

**(API)** You are provided with a list of candidate APIs. While not all may be correct, the one ranked at the top is usually the most accurate match. Your task is to use the correct APIs from the list when they apply. If the listed APIs are incorrect or do not fulfill the described functionality, you should suggest and use more appropriate alternatives. Below are the candidate APIs that may serve as replacements for the API  $\alpha$ , listed in descending order of relevance:

1. API Name:  $\alpha_1$ , Parameters:  $\vec{p}_1$
2. API Name:  $\alpha_2$ , Parameters:  $\vec{p}_2$
3. API Name:  $\alpha_3$ , Parameters:  $\vec{p}_3$

As a result, we obtain the LLM-generated code  $P_\alpha^{llm}$  for the sub-task  $(P_\alpha, \alpha)$ .

### 3.3 Migration-Action Generation

Given the LLM-generated code  $P_\alpha^{llm}$ , we construct the migration actions to transplant this code back into the original sliced program  $P_\alpha$ .

Algorithm 2 presents the overall procedure for the function  $\text{Actions}(P_\alpha, P_\alpha^{llm}, \alpha, L_{new})$ . This process involves two main steps: (1) identifying the migrated API in  $P_\alpha^{llm}$  and (2) post-processing the generated code.

**Algorithm 2** Migration-Action Generation (Actions)**Input:** Original sliced program  $P_\alpha$ , LLM-migrated code  $P_\alpha^{llm}$ , target API  $\alpha$ , new library  $L_{new}$ **Output:** A set of migration actions  $\mathcal{M}$ 

```

1:  $\mathcal{M} \leftarrow \emptyset$ 
   /* Identifying the Migrated API Statement */
2:  $\Delta \leftarrow GumTree(P_\alpha, P_\alpha^{llm})$ 
3:  $S_\alpha \leftarrow CallStmt(P_\alpha, \alpha)$ 
4:  $S'_\alpha \leftarrow MatchStmt(P_\alpha^{llm}, \Delta, S_\alpha)$ 
5: if  $S'_\alpha = \perp$  then
6:   return {Delete  $S_\alpha$ }
   /* Transplanting the Migrated Code */
7:  $P'_\alpha \leftarrow P_\alpha[S_\alpha \mapsto S'_\alpha]$ 
8: if  $S_\alpha = (y = E) \wedge S'_\alpha = (x = E') \wedge (x \neq y)$  then
9:    $S'_\alpha \leftarrow S'_\alpha[x \mapsto y]$ 
10:  $\theta \leftarrow UndefVars(P'_\alpha)$ 
11:  $\lambda \leftarrow \emptyset$ 
12: while  $\theta \neq \emptyset$  do
13:    $\lambda \leftarrow \lambda \cup \theta$ 
14:   for  $x \in \theta$  do
15:      $S_d \leftarrow GetDef(P_\alpha^{llm}, x)$ 
16:     if  $S_d = (x = E)$  then
17:        $S_m \leftarrow MatchStmt(P_\alpha, \Delta^{-1}, S_d)$ 
18:       if  $S_m = (y = E')$  then
19:          $\mathcal{M} \leftarrow \mathcal{M} \cup \{\text{Change}(x, y)\}$ 
20:          $S'_\alpha \leftarrow S'_\alpha[x \mapsto y]$ 
21:       else
22:          $S'_\alpha \leftarrow S_d; S'_\alpha$ 
23:       else if  $S_d = (\text{from } path \text{ import } x)$  then
24:          $path' \leftarrow ModifyPath(L_{new}, path, x)$ 
25:          $S'_\alpha \leftarrow (\text{from } path' \text{ import } x); S'_\alpha$ 
26:    $P'_\alpha \leftarrow P_\alpha[S_\alpha \mapsto S'_\alpha]$ 
27:    $\theta \leftarrow UndefVars(P'_\alpha) \setminus \lambda$ 
28: return  $\mathcal{M} \cup \{\text{Modify}(S_\alpha, S'_\alpha)\}$ 

```

3.3.1 *Identifying the Migrated API Statement.* To identify the migrated statement in the LLM-generated code, at line 2, we begin by applying a tree difference algorithm [15]

$$GumTree : Program \times Program \rightarrow ProgramMap$$

whose output  $\Delta \in ProgramMap = (Stmt + Expr) \rightarrow (Stmt + Expr + \perp)$  associates each statement or expression in the original program with its corresponding transformed statement or expression, or returns  $\perp$  indicating deletion in the modified program.

Next, we use a helper function  $CallStmt(P_\alpha, \alpha)$  that returns the statement in  $P_\alpha$  where the API  $\alpha$  is invoked. Using this function, we identify the API call statement  $S_\alpha$  in the original sliced program  $P_\alpha$  (line 3).

Let  $S'_\alpha$  be  $\Delta[S_\alpha]$  and assume  $S'_\alpha \neq \perp$ . In this case, we can determine that the call statement  $S_\alpha$  has been modified to  $S'_\alpha$  in the LLM-migrated code. However, in some cases, the standard

tree-differencing algorithm fails to detect the migrated statement and returns  $\perp$  due to substantial changes in usage patterns or surrounding code introduced by the LLM. To address this limitation, we define and use a function  $MatchStmt : Program \times ProgramMap \times Stmt \rightarrow (Stmt + \perp)$ :

$$MatchStmt(P, \Delta, S) = \begin{cases} \Delta[S] & \cdots \Delta[S] \neq \perp \\ Vote(P, \Delta, S) & \cdots \Delta[S] = \perp \end{cases}$$

When *GumTree* determines that the statement was removed ( $\Delta[S] = \perp$ ), we utilize an alternative fallback mechanism  $Vote(P, \Delta, S)$  to heuristically identify a likely migrated code instead of simply concluding that the statement was removed:

$$Vote(P, \Delta, S) = \arg \max_{S' \in Stmts(P)} Count(\Delta, S, S')$$

where  $Stmts(P)$  denotes the statement set in  $P$ . In the case of ties,  $\perp$  is returned (i.e., no suitable candidate found).

The function  $Count(\Delta, S, S')$  counts how many times sub-expressions in the original statement  $S$  are matched with sub-expressions in the candidate statement  $S'$ :

$$Count(\Delta, S, S') = |\{E \in Exprs(S) \mid \Delta[E] \in Exprs(S')\}|$$

where  $Exprs(S)$  returns the set of sub-expressions in  $S$ .

After applying the function  $MatchStmt$ , if the result is still  $\perp$  (line 5), we treat this as evidence that the API has been removed in the LLM-generated code and generate a Delete action (line 6).

**3.3.2 Transplanting the Migrated Code.** Otherwise, we perform post-processing to transplant the statement  $S'_\alpha$  back into the original sliced program  $P_\alpha$  without introducing errors. Post-processing is needed for two main reasons: naive transplantation often produces (1) undefined variables and (2) incorrectly imported paths or libraries.

In particular, undefined variables may arise when the LLM-generated code  $P_\alpha^{llm}$  is directly transplanted into the original sliced program  $P_\alpha$  without proper context integration. We identified two common cases where undefined variables may arise: (1) Variable Name Changes: Variable names in the LLM-generated code may differ from those in the original code. (2) Missing Context: To use the new API, it is necessary to extract additional context from the LLM-generated code  $P_\alpha^{llm}$ .

Before describing the post-processing algorithm, we define two helper functions. The first,  $UndefVars(P)$ , returns the set of undefined variables in program  $P$ . The second,  $GetDef(P, x)$ , returns either an assignment statement ( $x = E$ ) or an import statement (from *path* import  $x$ ) that defines  $x$  in program  $P$  (For presentation simplicity, we assume a single definition point for each variable).

**(1) Variable Renaming.** To maintain consistency with the original program even when the LLM changes variable names, such renamed variables must be mapped back to their original names. We assume that the LLM has renamed the variable  $y$  in the original program to  $x$  in the LLM-generated code if the variable  $x$  holds the same or a similar value as the variable  $y$ . To identify such renamed variables, we first check whether  $S_\alpha$  and  $S'_\alpha$  assign their return values to different variables,  $y$  and  $x$ , respectively (line 8). If so, we update the LLM-generated code to replace  $x$  with  $y$  (line 9).

We then replace the original API call statement  $S_\alpha$  with the LLM-generated migration statement  $S'_\alpha$  (line 7) and check whether it introduces any new undefined variables (line 10). If any undefined variables exist, we first extract the statement  $S_d$  that defines the variable  $x$  in the LLM-generated code  $P_\alpha^{llm}$  (line 15). If a set of undefined variables  $\theta$  is non-empty, for each variable  $x \in \theta$ , we first extract the statement  $S_d$  that defines the variable  $x$  in the LLM-generated code  $P_\alpha^{llm}$  (line 15). If  $S_d$  is an assignment of the form  $x = E$  (line 16), we search for a statement in the original sliced code

$P_\alpha$  that closely matches  $x = E$  using the inverse map  $\Delta^{-1}$  and the *MatchStmt* function (line 17). If the statement  $x = E$  is matched with  $y = E'$  (line 18), we infer that  $x$  has been renamed from  $y$ , and update the LLM-generated code to replace  $x$  with  $y$  (lines 19-20).

**(2) Context Augmentation.** Otherwise, if no matching variable is found in the original code, we treat the undefined variable  $x$  as a newly declared variable required by the migrated API. In such cases, we extract the corresponding assignment statement ( $x = E$ ) from the LLM-generated code and insert it into the original program. Specifically, the assignment ( $x = E$ ) is placed immediately before the migrated API call statement  $S'_\alpha$ , ensuring that  $x$  is properly defined before use (line 22).

If the statement  $S_d$  is an import statement (line 23), it indicates that the undefined variable  $x$  corresponds to an API name and that the relevant import must be transplanted into the original program. Rather than directly inserting  $S_d$ , we first check whether the import statement is correct, which ensures that the API is imported from the appropriate module path in the target library  $L_{\text{new}}$  before including it in the program. We define the function *ModifyPath* (line 24) to check and, if necessary, correct the import path in the statement  $S_d$ : If the tuple  $(x, \text{path}, \_)$  exists in the new library  $L_{\text{new}}$ , we treat the original path  $\text{path}$  as correct and *ModifyPath* simply returns  $\text{path}$ . Otherwise, it computes the set  $\Psi_x$  of all possible paths in  $L_{\text{new}}$  that define API  $x$ , i.e.,  $\Psi_x = \{\text{path} \mid (x, \text{path}, \_) \in L_{\text{new}}\}$ , and selects the path in  $\Psi_x$  that is most similar to the original path  $\text{path}$  according to the string similarity measure:

$$\text{ModifyPath}(L_{\text{new}}, \text{path}, x) = \arg \max_{\text{path}' \in \Psi_x} \text{StrSim}(\text{path}, \text{path}').$$

Finally, we place the modified import statement (from  $\text{path}'$  import  $x$ ) right before the migrated API call statement  $S'_\alpha$  (line 25).

### 3.4 Applying Migration Actions

To obtain the final migrated program  $P_{\text{new}}$ , we apply all migration actions in  $\mathcal{M}$  to the original program  $P_{\text{old}}$ , i.e.,  $P_{\text{new}} = \mathcal{M}(P_{\text{old}})$  (line 7 of Algorithm 1). Since migration actions may conflict with one another, such as a deletion action *Delete*( $S$ ) and a modification action *Modify*( $S, S'$ ) targeting the same statement, we apply the actions in the following order: *Delete*, *Modify*, and *Change*. Applying migration actions in this order ensures consistency. In particular, variable renaming is applied globally after the structure of the program is finalized by *Delete* and *Modify*.

### 3.5 Handling Various Migration Scenarios

PIG is compositionally designed to handle various API mapping scenarios beyond simple one-to-one correspondences. As described in Section 3.3, our modular pipeline explicitly supports one-to-zero and one-to-many mappings, and implicitly accommodates many-to-one and n-to-m mappings through sub-task decomposition and post-processing. For one-to-zero mappings, the *MatchStmt* function returns  $\perp$  when no corresponding statement is found in the LLM-generated code. In such cases, PIG issues a *Delete* action to safely remove the original API call while preserving correctness. For one-to-many mappings, PIG leverages context augmentation: although a sub-task begins with a single API call, the LLM may generate multiple related statements, which are integrated during transplantation. For many-to-one and n-to-m mappings, PIG decomposes multi-API migrations into smaller subtasks (one-to-one, one-to-zero, or one-to-many), then combines the results through shared context and post-processing into coherent transformations. The dataset used in our evaluation includes all such cases.

## 4 Evaluation

In this section, we evaluate the performance of PIG to answer the following research questions:

Table 1. The result of the evaluation of baseline configuration from prior work [23] and PiG on library migration tasks. # Correct APIs: the number of APIs that were successfully migrated. # Files: the number of files in which the proportion of successfully completed API calls exceeds specified thresholds (e.g., 100%,  $\geq 75\%$ ,  $\geq 50\%$ ,  $\geq 25\%$ ).

Settings		# Correct APIs (Total: 364)	# Files (Total: 69)			
Model	Type		Percentage of correct APIs			
			100%	$\geq 75\%$	$\geq 50\%$	$\geq 25\%$
<b>qwen2:7b</b>	BASELINE	81	3	6	17	32
	PiG	( $\uparrow 103.7\%$ ) 165	( $\uparrow 133.3\%$ ) 7	( $\uparrow 116.7\%$ ) 13	( $\uparrow 111.8\%$ ) 36	( $\uparrow 65.6\%$ ) 53
<b>gemma2:9b</b>	BASELINE	111	7	15	25	34
	PiG	( $\uparrow 78.4\%$ ) 198	( $\uparrow 114.3\%$ ) 15	( $\uparrow 53.3\%$ ) 23	( $\uparrow 64.0\%$ ) 41	( $\uparrow 55.9\%$ ) 53
<b>llama3.1:8b</b>	BASELINE	106	8	13	27	36
	PiG	( $\uparrow 76.4\%$ ) 187	( $\uparrow 37.5\%$ ) 11	( $\uparrow 38.5\%$ ) 18	( $\uparrow 48.1\%$ ) 40	( $\uparrow 55.6\%$ ) 56
<b>qwen3:32b</b>	BASELINE	115	12	17	28	34
	PiG	( $\uparrow 85.2\%$ ) 213	( $\uparrow 41.7\%$ ) 17	( $\uparrow 41.2\%$ ) 24	( $\uparrow 60.7\%$ ) 45	( $\uparrow 85.3\%$ ) 63
<b>gemma3:27b</b>	BASELINE	141	7	16	30	40
	PiG	( $\uparrow 47.5\%$ ) 208	( $\uparrow 157.1\%$ ) 18	( $\uparrow 62.5\%$ ) 26	( $\uparrow 50.0\%$ ) 45	( $\uparrow 45.0\%$ ) 58
<b>deepseek-r1:32b</b>	BASELINE	154	12	21	34	45
	PiG	( $\uparrow 37.7\%$ ) 212	( $\uparrow 41.7\%$ ) 17	( $\uparrow 28.6\%$ ) 27	( $\uparrow 29.4\%$ ) 44	( $\uparrow 35.6\%$ ) 61
<b>gpt-oss:20b</b>	BASELINE	221	24	34	46	51
	PiG	( $\uparrow 10.0\%$ ) 243	( $\uparrow 12.5\%$ ) 27	( $\uparrow 8.8\%$ ) 37	( $\uparrow 21.7\%$ ) 56	( $\uparrow 19.6\%$ ) 61
<b>Average</b>	BASELINE	132.7	10.4	17.4	29.6	38.9
	PiG	( $\uparrow 53.5\%$ ) 203.7	( $\uparrow 53.4\%$ ) 16.0	( $\uparrow 37.7\%$ ) 24.0	( $\uparrow 48.3\%$ ) 43.9	( $\uparrow 48.9\%$ ) 57.9

- (1) **Effectiveness:** How effective is PiG in automating library migration tasks?
- (2) **Ablation Study:** How does each component of PiG contribute to the overall performance?
- (3) **Generality:** Can PiG work well to new, unseen data? (Data Leakage)
- (4) **Limitations:** What are the limitations of PiG?

**Implementation.** We implemented PiG in approximately 15,000 lines of Python code (version 3.10.10). We used definition-use chains to implement the code slicing and utilized the `difflib` library to implement the string similarity algorithm. To effectively identify the migrated API statement, we modified the `gumtree` library to consider all sub-expressions. Additionally, we checked whether the identified statement contains an API from the new library  $L_{\text{new}}$ . When we collected the import statements to modify the import path, we performed validation to ensure that they do not cause argument or circular import errors. In addition, we designed the variable renaming to handle cases where duplicate names exist in the code.

**Model Selection.** We employed seven open-source LLMs for our experiments. Four of these are recent models (gpt-oss:20b [35], deepseek-r1:32b [13], gemma3:27b [39], and qwen3:32b [46]), while the other three are older, smaller models (llama3.1:8b [18], gemma2:9b [40], and qwen2:7b [47]). We deliberately restricted our evaluation to open-source models to allow the research community to replicate and extend our findings without access constraints.

We used the Ollama framework [33] to run the models on a system equipped with an NVIDIA RTX A6000 GPU (48GB VRAM) and 128GB RAM. We set the context window to 4,096 tokens and the temperature to 1.0.

## 4.1 Effectiveness & Ablation Study

**4.1.1 Setup.** In this section, we describe the experimental setup for evaluating the effectiveness of PiG and its individual components.

**Benchmarks.** In this study, we utilized the PYMIGBENCH2.0 [21], a dataset designed for library migration tasks in Python. Due to the practical challenges associated with evaluating migration outputs, we selected a representative subset for analysis. The primary challenge is the lack of test cases for many migration instances, which hinders automated validation. Furthermore, it is not feasible to rely solely on syntactic similarity or exact matches: even when the semantics of the original and migrated code are equivalent, the migrated version may invoke different APIs or exhibit structural variations. Consequently, manual verification was necessary to ensure evaluation accuracy.

Therefore, to keep the manual verification process manageable, we selected, for each unique library pair, the file with the largest number of API migration tasks—thus arguably the most challenging cases—among those with multiple migrations. This resulted in 69 files, comprising a total of 364 target APIs for evaluation.

**Success Criteria.** We evaluated the success of each migration by manually verifying the semantic equivalence between the original and migrated code. We utilized the ground-truth (i.e., the developer-used API) or library documentation as a reference point for this verification process. To minimize human error, we implemented a cross-check procedure among authors. When all evaluators reached the same conclusion, the label was assigned accordingly. In cases of disagreement, the migration was labeled as incorrect.

**Baseline.** Among existing works [1, 23], the zero-shot prompt from Islam et al. [23] was chosen as the baseline (BASELINE). We opted for this prompt because it is more comprehensive and encompasses the content of Almeida et al. [1]. The selected prompt is presented in a highlighted box below.

### Baseline 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:

**4.1.2 Effectiveness.** Table 1 presents the performance of PiG compared to baseline configurations on library migration tasks. At the API level, PiG achieved an average increase of 53.5% in successfully migrated APIs across all models. At the file level, it led to a 53.4% increase in fully correct files and a 48.3% increase in files with at least 50% correct migrations. Overall, PiG consistently outperformed BASELINE, demonstrating its robustness and effectiveness in automating library migration tasks.

**4.1.3 Ablation Study.** To evaluate the contribution of each component of PiG, we conducted an ablation study by incrementally adding each component on top of the SLICING configuration:

- SLICING: Incorporates only subtask decomposition via slicing and prompt guidance to mitigate SEM, MIG, and MIN errors.
- API: SLICING with the plausible API candidates.
- MATCH: API with the statement matching technique.
- PiG: MATCH with the post-processing technique.

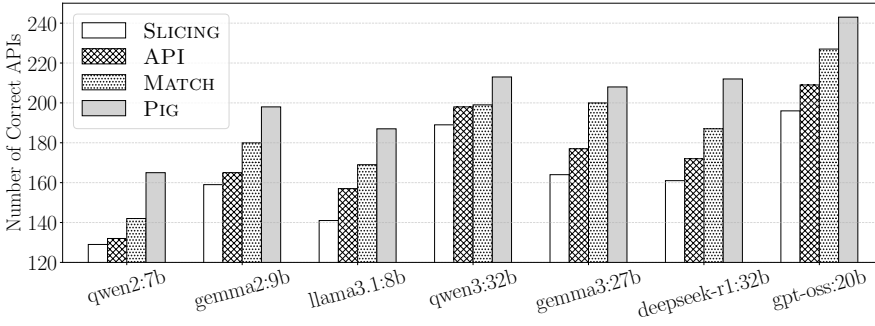


Fig. 6. Ablation study by incrementally adding each component of PIG on top of the SLICING model.

In the case of SLICING, we use a default query (without plausible API candidates) and apply a standard tree difference algorithm without any post-processing when identifying the migration-relevant statements.

Compared to SLICING, the inclusion of plausible API candidates (API) and statement matching (MATCH) led to average improvements of 6.2% and 14.6% in the number of successfully migrated API, respectively. The addition of a post-processing technique further enhanced the performance, resulting in a total average gain of 25.7% over SLICING. These observed stepwise gains demonstrate both the individual and combined contributions of each component in the PIG pipeline.

#### 4.2 Evaluating Generality on Post-Cutoff Commits

To assess the generality of PIG and address potential concerns regarding training data leakage, we curated a dataset from migration commits made after April 2025, which is beyond the training cut-off date of all evaluated models. For the 69 library pairs considered in the previous section, we searched GitHub commit messages across all public repositories using the GitHub search engine. We targeted commits whose messages mentioned both library names alongside migration-related keywords (e.g., migrate, move, replace). We then manually inspected whether each candidate commit represents a library migration, and when confirmed, collected the associated changed files. In total, we gathered 22 migrated files, encompassing 81 target APIs across 22 library pairs.

Table 2 presents the evaluation results of BASELINE and PIG on this post-cutoff dataset. PIG consistently outperformed BASELINE across all models, achieving an average improvement of 38.3% in successfully migrated APIs. Notably, it improved the performance of the strongest model, gpt-oss, by 17.0%. These results demonstrate that PIG remains effective even on migration instances that were not part of the training data, supporting its generalizability and robustness to data leakage concerns.

#### 4.3 Limitations and Future Work

The evaluation also revealed limitations of PIG, pointing to potential directions for future work. To better understand these limitations, we conducted a qualitative analysis of failure cases. Specifically, we examined all 44 cases where every model failed under the PIG configuration. For each case, we analyzed outputs from all seven models, resulting in a total of 308 instances (44 cases  $\times$  7 models) for detailed evaluation.

**Incorrect APIs in LLM Output.** Although PIG provides plausible API candidates to the LLM, 106 cases of the responses still use APIs incorrectly. Figure 7a illustrates an example such a case: despite PIG suggests the correct API candidate `verifier`, the LLM generates `set_verifier`, an incorrect

Table 2. Evaluation results on 81 target APIs collected after April 2025.

Model	qwen2	gemma2	llama3.1	qwen3	gemma3	deepseek-r1	gpt-oss	Average
BASELINE	24	36	37	50	38	52	53	41.4
PIG	57	59	50	58	59	56	62	57.3
% Increase	▲137.5%	▲63.9%	▲35.1%	▲16.0%	▲55.3%	▲7.7%	▲17.0%	▲38.3%

API that does not exist in the target library. This issue arises because the LLM does not always adhere to the provided candidates, even when they are correct.

In the remaining 85 cases, PIG fails to provide the correct API candidate to the LLM. These cases typically involve source and target APIs with significantly different syntax or naming conventions, e.g., `add_argument` vs. `option`, or `encrypt` vs. `hashpw`. Currently, PIG generates API candidates based on simple heuristics, such as name similarity and type compatibility. This approach may not be sufficient to capture complex relationships between APIs, when they have different names or when the target library has a different design paradigm, leading to one-to-zero mapping scenarios. As future work, we plan to enhance the API candidate generation method to better handle such challenging cases. See Section 5 for related work on API mapping techniques.

**Statement Matching.** In 80 cases, PIG fails to correctly identify the migrated API statement. For example, Figure 7b shows a migration example from the `attributes` library to the `attr` library. In this case, the statement at line 3 in the left code is expected to the statement at line 5 in the right code. However, PIG fails to match them due to significant syntactic and sub-expression differences. To handle such cases, future work could explore more sophisticated, semantics-aware approaches beyond AST-based comparisons. For example, incorporating string-level matching to align elements like the literal `'num'` on the left code (line 3) with the variable `num` on the right code (line 5).

**Extra Errors.** The remaining 37 cases involve ill-formed or incomplete code, which hinders AST-based matching and complicates the accurate transplantation of migrated code. In such scenarios, remedial mechanisms, such as re-querying the LLM to generate a valid output, may be necessary.

## 5 Related Work

**Traditional Approaches.** Prior work on automating library migration has primarily relied on code examples or external documentation. Ni et al. [32] proposed SOAR, a synthesis-based technique that leverages runtime feedback and documentation to support migration between Python ML libraries. Xu et al. [45] infer migration edits by generalizing patterns from example-based transformations. Other examples include techniques that mine code change patterns from migration commits, either to analyze API differences through static analysis [2, 4, 41, 45], or to extract migration rules using machine learning techniques [3, 14, 16, 27]. However, these approaches are typically limited by their reliance on curated migration examples or documentation. MELT [36] also mines code change patterns from pull requests from library repositories. Yet, they are primarily designed for intra-library migrations (i.e., version updates) and do not easily scale to the more complex inter-library migrations.

**LLM-based Approaches.** More recently, researchers have explored the use of LLMs to overcome the limitations of traditional migration techniques. Zhou et al. [50] proposed a hybrid technique that supplements LLMs with small API mapping models learned from prior migration examples. However, this approach requires the prior collection of migration examples for each specific pair of libraries, which restricts its applicability to unseen library pairs. Islam et al. [23] conducted an

<pre> 1 import oauth2 2 c = oauth2.Client() 3 v = ... 4 5 6 # should be migrated 7 c.set_verifier(v) </pre>	<pre> 1 import oauth1 2 c = oauth1.Client() 3 v = ... 4 # Incorrect LLM API 5 c.set_verifier(v) 6 # correct API 7 c.verifier = v </pre>	<pre> 1 import attributes 2 3 @attributes(['num']) 4 class Session(): 5 ... </pre>	<pre> 1 import attr 2 3 4 class Session: 5     num = attr.ib() </pre>
---	---	--	---

(a) The LLM uses an incorrect API statement at line 5 which does not exist in the target library. The expected API statement is `c.verifier = v`.

(b) The statement at line 3 in the left code is expected to be the statement at line 5 in the right code.

Fig. 7. Examples of failure cases in PtG. The first example shows the case where the LLM uses an incorrect API, while the second example shows the case where the LLM fails to match the migrated API statement.

empirical study on the effectiveness of LLMs (e.g., Llama 3.1 [18] and GPT-4o [34]) in automating Python library migrations, providing a detailed analysis of the correctness of the LLM-based migration in various aspects. However, those insights were not translated into a systematic or automated solution to mitigate the migration failures. Almeida et al. [1] also investigated LLM-based migration, introducing several prompting strategies to improve LLM performance. However, their study was limited to a single intra-library migration (from SQLAlchemy version 1 to 2), and the approach is difficult to automate due to its reliance on task-specific prior knowledge. In contrast, PtG systematically analyzes common failure patterns in LLM-based migration and introduces a modular, automated pipeline that integrates context slicing, prompt tuning, and code transplantation. This design enables more controlled and generalizable use of LLMs for diverse multi-API migration tasks.

**Automated API Mapping.** Automated API mapping is a well-studied problem, with a range of techniques proposed to identify corresponding APIs across libraries. Existing approaches can be broadly categorized into two categories: rule-based methods [12, 20, 42] and learning-based methods. Recently, learning-based approaches have gained traction for their ability to capture complex relationships between APIs. Several studies [9, 10, 17, 30, 49, 50] leverage code examples to identify the API mapping relationships through statistical and machine learning techniques, while others exploit text similarity in documentation to identify equivalent APIs [28, 31, 48].

However, most prior research has focused on the Java language, and there remains a lack of off-the-shelf API mapping tools for Python. For example, Zhou et al. [50]’s evaluation was conducted only on cross-library (Java libraries) and cross-language (Java to C#) scenarios. In addition, in the Python ecosystem, extracting migration-related data still demands extensive manual effort [22], and official library documentation is often sparse or inconsistent. These factors limit the applicability of existing techniques that rely heavily on high-quality documentation or curated datasets. To address this gap, PtG introduces a prompting strategy designed to significantly improve LLM-based API mapping in a zero-shot setting and also incorporates a systematic post-processing methodology, eliminating the need for pre-collected code examples or external documentations.

**LLMs for Software Engineering.** LLMs are increasingly used across diverse software engineering tasks, including refactoring [11, 38], automated program repair [43, 44], test generation [5, 43], code translation [29], etc. We complement these efforts by demonstrating that LLMs can also be effectively leveraged for an important class of software maintenance tasks: automated library migration. Furthermore, recent studies explore the use of LLM-based agents to tackle complex software engineering tasks by identifying optimal tool integration to maximize their effectiveness [7, 8, 37].

Our work aligns with this emerging direction by identifying failure patterns of LLMs and uncovering specific bottlenecks in the migration process. By pinpointing these limitations, we expect our findings to serve as a foundation for building more robust LLM agents for library migration in the future.

## 6 Conclusion

As software ecosystems evolve, developers often need to switch libraries to adopt better-supported, more efficient, or more secure alternatives. However, such migration tasks are challenging, as they require a deep understanding of the APIs and the ability to write semantically equivalent code. To address this challenge, we presented Pig, a new approach for automating Python library migration using Large Language Models (LLMs). Pig decomposes the migration task into API-level sub-tasks, guides LLMs with plausible candidates and error-informed prompts, and applies post-processing techniques to improve the accuracy and reliability of LLM-based migrations. Our evaluation on 364 real-world API-level migration tasks demonstrates that Pig outperforms the baseline by 53.5%, in terms of correctly migrated APIs.

## Acknowledgments

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains, 30%) and by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2026-RS-2020-II201819, 5%). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No.2021R1A5A1021944(20%), RS-2026-25470524(30%)). Gabin An and Hakjoo Oh are corresponding authors.

## Data Availability

The source code of our tool implementation, benchmarks we used, and experimental results are archived on Zenodo at [24] and also available at [https://github.com/kupl/pig\\_artifact](https://github.com/kupl/pig_artifact).

## References

- [1] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 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* (Barcelona, Spain) (ESEM '24). Association for Computing Machinery, New York, NY, USA, 427–433. doi:10.1145/3674805.3690746
- [2] Hussein Alrubaye and Mohamed Wiem Mkaouer. 2018. Automating the detection of third-party Java library migration at the function level.. In *CASCON*. 60–71.
- [3] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. 2020. Learning to recommend third-party library migration opportunities at the API level. *Applied Soft Computing* 90 (2020), 106140.
- [4] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. 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, 347–357.
- [5] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839
- [6] Paul E. Black. 2021. *Ratcliff/Obershelp pattern recognition*. <https://www.nist.gov/dads/HTML/ratcliffObershelp.html> Accessed: 2025-05-07.

- [7] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) (ICSE '25). IEEE Press, 2188–2200. doi:10.1109/ICSE55347.2025.00157
- [8] Islem Bouzenia and Michael Pradel. 2025. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA047 (June 2025), 23 pages. doi:10.1145/3728922
- [9] Nghi D. Q. Bui. 2019. Towards zero knowledge learning for cross language API mappings. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings* (Montreal, Quebec, Canada) (ICSE '19). IEEE Press, 123–125. doi:10.1109/ICSE-Companion.2019.00054
- [10] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) (ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 796–806. doi:10.1145/3338906.3338924
- [11] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2024. An Empirical Study on the Code Refactoring Capability of Large Language Models. arXiv:2411.02320 [cs.SE] <https://arxiv.org/abs/2411.02320>
- [12] Barthélemy Dagenais and Martin P. Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. In *2009 IEEE 31st International Conference on Software Engineering*. 599–602. doi:10.1109/ICSE.2009.5070565
- [13] DeepSeek-AI, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, et al. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. arXiv:2401.02954 [cs.CL] <https://arxiv.org/abs/2401.02954>
- [14] Juri Di Rocco, Phuong T. Nguyen, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2025. DeepMig: A transformer-based approach to support coupled library and code migrations. *Inf. Softw. Technol.* 177, C (Jan. 2025), 19 pages. doi:10.1016/j.infsof.2024.107588
- [15] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) (ASE '14). Association for Computing Machinery, New York, NY, USA, 313–324. doi:10.1145/2642937.2642982
- [16] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2020. Apimigrator: an api-usage migration tool for android apps. In *Proceedings of the IEEE/ACM 7th international conference on mobile software engineering and systems*. 77–80.
- [17] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. 2013. Inferring likely mappings between APIs. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) (ICSE '13). IEEE Press, 82–91.
- [18] Aaron Grattafiori, Abhimanyu Dubey, Angela Fan, Emily Dinan, Gabriel Synnaeve, Mike Lewis, Naman Goyal, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] <https://arxiv.org/abs/2407.21783>
- [19] Haiqiao Gu, Hao He, and Minghui Zhou. 2023. Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 627–638. doi:10.1109/SANER56733.2023.00064
- [20] J. Henkel and A. Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. 274–283. doi:10.1109/ICSE.2005.1553570
- [21] Mohayeminul Islam, Ajay Kumar Jha, Ildar Akhmetov, and Sarah Nadi. 2024. Characterizing Python Library Migrations. *Proc. ACM Softw. Eng.* 1, FSE, Article 5 (July 2024), 23 pages. doi:10.1145/3643731
- [22] Mohayeminul Islam, Ajay Kumar Jha, Sarah Nadi, and Ildar Akhmetov. 2023. PyMigBench: A Benchmark for Python Library Migration. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 511–515. doi:10.1109/MSR59073.2023.00075
- [23] Md Mohayeminul Islam, Ajay Kumar Jha, May Mahmoud, Ildar Akhmetov, and Sarah Nadi. 2025. Using LLMs for Library Migration. arXiv:2504.13272 [cs.SE] <https://arxiv.org/abs/2504.13272>
- [24] Miryeong Kang. 2026. Artifact Evaluation Package for "Pig: Leveraging Large Language Models for Python Library Migrations". doi:10.5281/zenodo.19480524
- [25] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.
- [26] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23, 1 (May 2017), 384–417. doi:10.1007/s10664-017-9521-5
- [27] Maxime Lamothe, Weiwei Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering* 48, 2 (2020), 417–431.
- [28] Yangyang Lu, Ge Li, Zelong Zhao, Linfeng Wen, and Zhi Jin. 2017. Learning to Infer API Mappings from API Documents. In *Knowledge Science, Engineering and Management*, Gang Li, Yong Ge, Zili Zhang, Zhi Jin, and Michael Blumenstein (Eds.). Springer International Publishing, Cham, 237–248.

- [29] Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. 2024. Bridging Gaps in LLM Code Translation: Reducing Errors with Call Graphs and Bridged Debuggers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 2448–2449. doi:10.1145/3691620.3695322
- [30] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) (ASE '14). Association for Computing Machinery, New York, NY, USA, 457–468. doi:10.1145/2642937.2643010
- [31] Thanh Nguyen, Ngoc Tran, Hung Phan, Trong Nguyen, Linh Truong, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2018. Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 551–562. doi:10.1145/3236024.3236036
- [32] Ansong Ni, Daniel Ramos, Aidan Z.H. Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2021. SOAR: A Synthesis Approach for Data Science API Refactoring. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) (ICSE '21). IEEE Press, 112–124. doi:10.1109/ICSE43902.2021.00023
- [33] Ollama. 2023. Ollama GitHub Repository. <https://ollama.com>.
- [34] OpenAI. 2025. GPT-4o. <https://platform.openai.com/docs/models/gpt-4o>.
- [35] OpenAI. 2025. OpenAI gpt-oss-20b. <https://platform.openai.com/docs/models/gpt-oss-20b>.
- [36] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2024. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) (ASE '23). IEEE Press, 1516–1528. doi:10.1109/ASE56229.2023.00117
- [37] Devjeet Roy, Xuchao Zhang, Rashi Bhawe, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring LLM-Based Agents for Root Cause Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 208–219. doi:10.1145/3663529.3663841
- [38] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring Programs Using Large Language Models with Few-Shot Examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, Los Alamitos, CA, USA, 151–160. doi:10.1109/APSEC60848.2023.00025
- [39] Gemma Team, Aishwarya Kamath, Johan Ferret, et al. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] <https://arxiv.org/abs/2503.19786>
- [40] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Thomas Mesnard, Léonard Hussenot, Johan Ferret, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving Open Language Models at a Practical Size. arXiv:2408.00118 [cs.CL] <https://arxiv.org/abs/2408.00118>
- [41] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2013. Automatic discovery of function mappings between similar libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 192–201.
- [42] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) (ICSE '10). Association for Computing Machinery, New York, NY, USA, 325–334. doi:10.1145/1806799.1806848
- [43] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494. doi:10.1109/ICSE48619.2023.00129
- [44] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for \$0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 819–831.
- [45] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 335–346. doi:10.1109/ICPC.2019.00052
- [46] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] <https://arxiv.org/abs/2505.09388>
- [47] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, et al. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] <https://arxiv.org/abs/2407.10671>
- [48] Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. 2020. Deep-Diving into Documentation to Develop Improved Java-to-Swift API Mapping. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) (ICPC '20). Association for Computing Machinery, New York, NY, USA,

106–116. doi:10.1145/3387904.3389282

- [49] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) (*ICSE '10*). Association for Computing Machinery, New York, NY, USA, 195–204. doi:10.1145/1806799.1806831
- [50] Bingzhe Zhou, Xinying Wang, Shengbin Xu, Yuan Yao, Minxue Pan, Feng Xu, and Xiaoxing Ma. 2023. Hybrid API Migration: A Marriage of Small API Mapping Models and Large Language Models. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware* (Hangzhou, China) (*Internetware '23*). Association for Computing Machinery, New York, NY, USA, 12–21. doi:10.1145/3609437.3609466

Received 2025-09-12; accepted 2025-12-22