Characterizing Python Library Migrations

MOHAYEMINUL ISLAM, University of Alberta, Canada
AJAY KUMAR JHA, North Dakota State University, USA
ILDAR AKHMETOV, Northeastern University, Canada
SARAH NADI, University of Alberta, Canada

Developers heavily rely on Application Programming Interfaces (APIs) from libraries to build their software. As software evolves, developers may need to replace the used libraries with alternate libraries, a process known as *library migration*. Doing this manually can be tedious, time-consuming, and prone to errors. Automated migration techniques can help alleviate some of this burden. However, designing effective automated migration techniques requires understanding the types of code changes required to transform client code that used the old library to the new library. This paper contributes an empirical study that provides a holistic view of Python library migrations, both in terms of the code changes required in a migration and the typical development effort involved. We manually label 3,096 migration-related code changes in 335 Python library migrations from 311 client repositories spanning 141 library pairs from 35 domains. Based on our labeled data, we derive a taxonomy for describing migration-related code changes, PyMigTax. Leveraging PyMigTax and our labeled data, we investigate various characteristics of Python library migrations, such as the types of program elements and properties of API mappings, the combinations of types of migration-related code changes in a migration, and the typical development effort required for a migration. Our findings highlight various potential shortcomings of current library migration tools. For example, we find that 40% of library pairs have API mappings that involve non-function program elements, while most library migration techniques typically assume that function calls from the source library will map into (one or more) function calls from the target library. As an approximation for the development effort involved, we find that, on average, a developer needs to learn about 4 APIs and 2 API mappings to perform a migration, and change 8 lines of code. However, we also found cases of migrations that involve up to 43 unique APIs, 22 API mappings, and 758 lines of code, making them harder to manually implement. Overall, our contributions provide the necessary knowledge and foundations for developing automated Python library migration techniques. We make all data and scripts related to this study publicly available at https://doi.org/10.6084/m9.figshare.24216858.v2.

Additional Key Words and Phrases: Python, library migration, code transformation, development effort

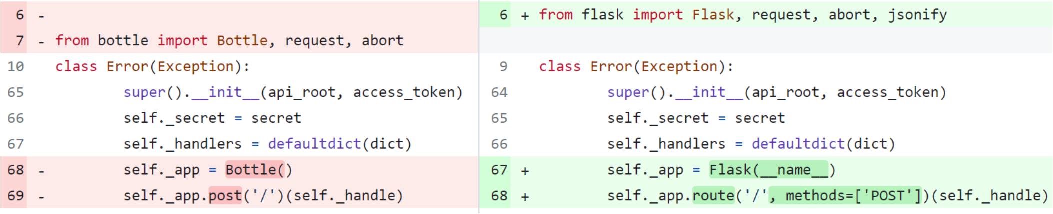
ACM Reference Format:
Mohayeminul Islam, Ajay Kumar Jha, Ildar Akhmetov, and Sarah Nadi. 2018. Characterizing Python Library Migrations. In . ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Modern software development heavily relies on third-party libraries [1, 11, 56], as they can enhance developer efficiency [43] and improve the reliability and maintainability of software systems [58]. However, the libraries that an application depends on may become obsolete over time [29, 49, 55]. Libraries with vulnerabilities or bugs can also adversely impact the applications that use them [20, 38]. Moreover, developers might discover newer, better-performing, or easier-to-use libraries [29, 33, 40]. In all these situations, developers often need to replace a currently used library with an alternative one, a process commonly referred to as *library migration*.

The library migration process typically involves finding which APIs from the new library can replace the used APIs from the old library (*API mapping*), and updating all existing code that used the old library's API to now use the new library's API while ensuring no change in the software's behavior (*client code transformation*). This is a time-consuming and error-prone task that developers

Preprint, January 28, 2024, Canada
2018. ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX



```
-
- from bottle import Bottle, request, abort
  class Error(Exception):
          super().__init__(api_root, access_token)
          self._secret = secret
          self._handlers = defaultdict(dict)
-         self._app = Bottle()
-         self._app.post('/')(self._handle)
```

```
+ from flask import Flask, request, abort, jsonify

  class Error(Exception):
          super().__init__(api_root, access_token)
          self._secret = secret
          self._handlers = defaultdict(dict)
+         self._app = Flask(__name__)
+         self._app.route('/', methods=['POST'])(self._handle)
```

Fig. 1. *bottle* to *flask* migration extracted/adapted from *cqmoe/python-cqhttp#f9f083ec*.

often dread [38], especially in large codebases with pervasive use of the original library. Therefore, migration techniques that automate this entire process can save developers time and effort.

With the exception of a few techniques [45], most of the existing library migration literature targets Java libraries [5, 8, 15, 53, 59]. Regardless of the supported language, most of these techniques stop at the API mapping stage [5, 8, 15, 53, 59] and/or implicitly assume that function calls from the source library map to (one or more) function calls in the target library [5, 8, 15, 45, 53, 59]. When implementing client code transformation, researchers may also leverage certain aspects of the target library domain to implement their technique, e.g., the expressiveness of error messages in the data science domain [45]. However, designing general library migration support techniques requires a deep and systematic understanding of the required migration-related code changes.

Figure 1 presents a Python code snippet migrated from the web application framework *bottle* (left side) to an analogous library, *flask* (right side). An API mapping technique may suggest replacing the `post()` function with the `route()` function. While helpful, the information is incomplete, because successfully transforming the shown client code to use the new library requires adding the `methods` argument with the value `['POST']` to the `route()` call.

Overall, there is no systematic knowledge of all types of API mappings and code changes for successful client code transformation from a source to a target library. For example, some analogous libraries may have similar APIs, requiring minimal code modifications during migration. Conversely, others may have considerable differences, requiring extensive modifications. This raises several pertinent questions for the development of library migration tools and techniques. *What are the common types of code changes that developers typically perform to migrate from one library to another? Does a migration typically involve a single kind of code change or multiple related code changes? How difficult would it be to automate these changes?* Overall, *What are the characteristics of the code changes during library migrations?*

To the best of our knowledge, while there is a lot of literature on library migration, no study systematically characterizes migration-related code changes regardless of the targeted language. Given the Python's rising popularity and its expanding library set [22], we address this gap by conducting an empirical study to understand different types of code changes that happen during Python library migration in open-source repositories. We analyze real-world Python library migrations using two existing datasets, PyMigBench [31] and SALM [25], creating a new dataset PyMigBench-2.0. While these datasets contain identified migration commits, they do not (identify or) analyze the involved code changes. In this work, we identify migration-related code changes from the migrations in PyMigBench-2.0, label these code changes, and then build a taxonomy of code changes, PyMigTax. The resulting labeled migrations allow us to perform the empirical study and investigate the above questions; the answers to these questions can aid tool builders and researchers in developing Python library migration tools and techniques.

Our empirical study is based on PyMigBench-2.0, which includes 3,096 migration-related code changes in 335 Python library migrations from 311 client repositories spanning 141 library pairs from 35 domains. We find that 40% of library pairs have API mappings that involve program elements other than functions (e.g., attribute or decorator). We also find that a program element from the source library (e.g., a function call) may be replaced by a different type of program element from the target library (e.g., an attribute access). These results imply that some of the assumptions that current API mapping or client transformation tools make (e.g., that a function is always replaced by another function [5, 6, 45, 53]) may not always hold in all migration scenarios. Regardless of the type of program elements in an API mapping, 65% of API mappings involve program elements with *different* names, and 29% of function call to function call API mappings involve some form of argument transformation. Thus, it is essential for tools to support a variety of types of code changes. We also find that, on average (median), a migration includes 8 lines of code, 7 API instances, and 3 code changes, and a developer who has to manually migrate client code needs to learn 4 APIs and 2 API mappings. Thus, on one hand, the good news is that an automated tool may be able to automate the majority of migrations, given their simplicity and repetitiveness. On the other hand, such automation requires handling several types of API mappings and argument transformations in certain more complex migrations.

To summarize, this paper makes the following contributions:

(1) We create a benchmark of 335 manually verified Python library migrations, PyMigBench-2.0, by expanding the original PyMigBench [31] dataset with verified migrations from SALM [25].
(2) We manually label 3,096 migration-related code changes in the PyMigBench-2.0 dataset to build PyMigTax, a taxonomy of Python library migrations.
(3) We conduct an empirical study on PyMigBench-2.0 to characterize the nature of Python library migrations in terms of the required migration-related code changes and migration effort.

Researchers can use PyMigBench-2.0 to evaluate their library migration tools and techniques and use PyMigTax for accurately labeling supported library migration types. For example, one technique may only work for client code transformation that involves function mapping with no changes in arguments while another may be able to additionally handle argument changes and modifications to attribute access. By clearly stating supported operations, techniques can be compared systematically and fairly. Our findings, along with the discussed implications, provide insights into the characteristics of Python library migrations, highlighting the required migration support. All our data and analysis scripts are available on our online artifact page https://doi.org/10.6084/m9.figshare.24216858.v2.

2 BACKGROUND AND TERMINOLOGY

2.1 Terminology and Notation

Library migration entails replacing one library with another in a software application, with the old being the *source library* and the new being the *target library*. Similarly, *source API* and *target API* refer to the APIs of these respective libraries. A library pair is termed *analogous* if the two libraries provide similar functionality, allowing migration between them.

A commit in which migration occurs is a *migration commit*. While migrations can span multiple commits [6], the datasets used in this study (see Section 2.2) consider only single-commit migrations. Therefore, for simplicity, this paper equates a migration with a migration commit. Nonetheless, a single commit can encompass multiple migrations between different library pairs.

During library migration, developers modify the client code to replace source library APIs with analogous target APIs. We refer to these modifications a *migration-related code changes* or simply *code changes*. A migration can require adjustments in multiple files and locations within a file. A single code change denotes a minimal API replacement that is indivisible into further meaningful replacements. For example, in Figure 1, we observe three distinct code changes. The import on line 7 is replaced with the import in line 6, the function call `Bottle()` on line 68 is replaced by the function call to `Flask()` on line 67, and the function call `post()` on line 69 is replaced by a call to function `route()` on line 68. We use the notation <removed-lines>:<added-lines> to specify a code changes in a given diff; hence these three code changes are denoted by 7:6, 68:67, and 69:68. Since the same line number may appear in both the removed or added lines, we use −<line-number> and +<line-number> to denote removed and added lines, respectively.

By observing code changes (along with reading the libraries' documentations), we can infer *API mappings*, a term commonly used in the literature [16, 44], to indicate which API(s) from the target library perform the same functionality provided by the original API from the source library. From the changes in Figure 1, we can infer two API mappings `Bottle()` → `Flask()` and `post()` → `route()`. To successfully apply API mappings during client code transformation, developers may need to perform additional types of code changes. We refer to such additional code changes as *properties*. For example, the code change that applied the API mapping from `Bottle()` to `Flask()` in Figure 1 required the following properties: name change and argument addition.

2.2 Datasets

We provide the background details of the two datasets we use: SALM [25] and PyMigBench [31].

SALM. Gu et al. [25] analyzed self-admitted library migrations (SALM) in Java, JavaScript and Python repositories to understand the nature and frequency of library migrations. Their contribution includes a dataset of self-admitted Python library migrations which we refer to as *SALM* in this paper. Gu et al. use libraries.io [34] to retrieve 10,147 popular Python libraries, and GHTorrent [24] to retrieve 121,381 client repositories for these libraries. To identify self-admitted migrations, they use NLP-based heuristics to analyze the commit messages to identify commits that explicitly mention a migration between two libraries. They further filter out migrations between infrequent library pairs and finally include 5,805 library migrations between 640 library pairs in 5,061 commits.

PyMigBench. In our previous work [31], we built PyMigBench, a benchmark of Python library migrations and locations of code changes. We used SEART [19] to retrieve a list of 195,075 non-forked Python repositories with 10 or more stars. We first used various automated filtering to discard unlikely migrations, and then manually verified 1,244 migrations between library pairs that frequently appear in the migrations, resulting in 75 migrations with code changes between 34 analogous library pairs.

3 BUILDING PYMIGTAX

For our empirical study, we need a unified way to describe migration-related code changes. Thus, as our first step, we use the identified migrations from the two datasets mentioned in Section 2.2, PyMigBench and SALM, to derive a taxonomy of code changes, referred to as PyMigTax. We will now describe the steps undertaken to construct PyMigTax.

3.1 Data curation

In our previous work, we manually validated each migration that is part of PyMigBench, enabling us to use it as is in this work. However, SALM is an automatically constructed data set, which relies on heuristics to detect self-admitted migrations from commit messages. Thus, we first verify the correctness of the data SALM provides, and apply the following filtering and preprocessing steps.

We remove 4 duplicate migrations found in SALM and focus on third-party libraries (the ones available in PyPI [22]). There are 35 libraries in SALM not on PyPI, either due to misspellings or

Table 1. Datasets used in the study

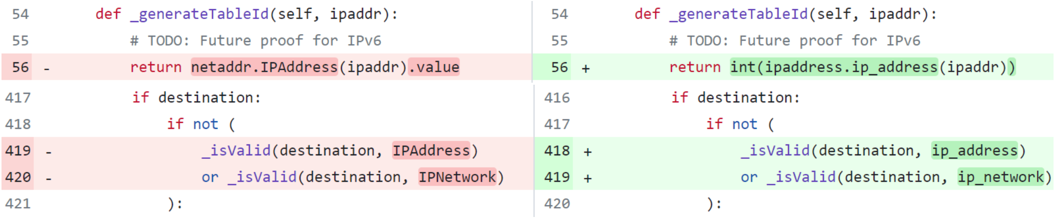
Round	Dataset	Migrations	Repos	Lib Pairs	Libs	Domains	κ PE	κ Cardinality	α Props
Full data after filtering	PyMigBench	75	57	34	55	11			
	SALM	1,559	1,291	199	265	25			
	All	1,634	1,338	224	296	36			
Initial	PyMigBench	75	57	34	55	11			
1	SALM	13	13	12	21	3	0.43	0.52	0.59
2	SALM	19	19	14	23	7	0.93	0.94	0.48
3	SALM	20	20	12	24	9	0.97	0.80	0.81
4	SALM	259	249	116	174	24			
Labeled data	PyMigBench	75	57	34	55	11			
	SALM	310	298	131	193	25			
	All	385	355	159	229	36			
Having code change	PyMigBench-2.0	335	311	141	208	35			

being system libraries, affecting 155 library pairs and resulting in the exclusion of 1,818 migrations. We observe some migrations in SALM between seemingly non-analogous library pairs, possibly due to its automated construction. An example includes migrations between *flask* (a web application framework) and *click* (CLI library). We identify 253 non-analogous library pairs in SALM using OpenAI's GPT-4 API [47] (prompt is available on the artifacts page) and exclude them and their corresponding migrations. This results in 1,559 self-admitted library migrations which we use. The first three rows of Table 1 summarize the two datasets after the above filtering. Given that some migrations overlap in both datasets, the *All* row does not directly sum the preceding two rows.

3.2 Building the initial version of PyMigTax

We build an initial version of PyMigTax based on the data available in PyMigBench because it specifies the line numbers of code changes for each recorded migration, unlike SALM. For each migration in PyMigBench, we manually analyze the modified Python files in the corresponding commit to understand and label the types of code changes, following an iterative open-coding [50] process as follows. The first author initially manually reviews code changes of 32 randomly selected migrations, recording the source and target APIs and how the replacement between them is happening in the migration-related code changes (e.g., *replace a function* and *add an argument*). Based on these notes, the first author creates a draft of the taxonomy that they discuss with two other authors, looking at the corresponding examples and discuss the structure and labels to use in the taxonomy. Based on the discussion, the first author revisits and refines the draft taxonomy, applying the new insights to label additional data before another round of discussion. This process continues until the three authors agree that the taxonomy accurately captures the observed code changes available in PyMigBench. At the end of this process, we identify the following three dimensions for labeling code changes:

Program elements. The program element are types of the APIs involved in the code change, such as function call and attribute. We further separate source and target program elements to indicate the types of source and target APIs, respectively. For example, in the change 56:56 in Figure 2, the function call `IPAddress` and the attribute `value` from the source library are replaced with the `ip_address` function call from the target library. Here, the source program elements are function call and attribute, with the target program element being function call.



```
54        def _generateTableId(self, ipaddr):                  54        def _generateTableId(self, ipaddr):
55            # TODO: Future proof for IPv6                      55            # TODO: Future proof for IPv6
56  -         return netaddr.IPAddress(ipaddr).value           56  +         return int(ipaddress.ip_address(ipaddr))
417           if destination:                                  416           if destination:
418               if not (                                     417               if not (
419 -                 _isValid(destination, IPAddress)         418 +                 _isValid(destination, ip_address)
420 -                 or _isValid(destination, IPNetwork)      419 +                 or _isValid(destination, ip_network)
421               ):                                           420               ):
```

Fig. 2. *netaddr* to *ipaddress* migration extracted/adapted from *ovirt/vdsm#6eef802*.

Cardinality. The numerical relationship between removed and added APIs in a code change. We use *many* to describe any quantity greater than one, as also used in the literature [8]. In the change 56:56 from Figure 2, two APIs are replaced with one, therefore, the cardinality is many-to-one.

Properties. Additional specific details that characterize the code change. Such properties emerged from our open coding process. For example, in Figure 2, the new code on Line 56 requires a cast to an integer (using `int()`), which we label as *output transformation*.

We use the notation $\textit{source program elements} \xrightarrow{\textit{cardinality}} \textit{target program elements} \mid \textit{properties}$ to label a code change using PyMigTax, often omitting the cardinality. For example, the change in line 56:56 in Figure 2 can be denoted as $\textit{function call, attribute} \rightarrow \textit{function call} \mid \textit{outTrans}$. For brevity, we do not show the initial version of PyMigTax in the paper, but include it in our artifact.

3.3 Extending PyMigTax

While PyMigBench contains manually verified data, its small size poses a risk of constraining our taxonomy, potentially overfitting it solely to the migrations observed in this dataset. To improve the generalizability of the taxonomy, we use the initial version of PyMigTax to label the code changes of a statistically representative sample of migrations in SALM. We follow an iterative process (outlined below) where we allow modifications to PyMigTax based on the new data and then finalize the taxonomy when we reach saturation.

3.3.1 Automatic Identification of Migration-related Changed Lines: SALM does not include the location of the migration-related code changes and might include commits with no actual changes, relying solely on self-admitted migration in commit messages. Thus, we first automatically identify lines that may potentially have code changes, focusing on those related to the source or target libraries in the self-admitted migration. However, this requires detecting library usage in the modified files. Although migrations in SALM are labeled with source and target library names, these names do not always match the import names; for instance, *pyyaml* uses the import name `yaml`. We use OpenAI's GPT-4 API [47] to retrieve the mappings between library names and their corresponding top-level import names (prompt is available on our artifacts page). Given the import names, we can now identify usages of the libraries in the changed code. Specifically, we use the `ast` module to find usages of the libraries. We then identify candidate migration lines as deleted lines where the source library was used or added lines where the target library is used. We find a total of 1,269 of the 1,559 migrations in SALM with at least one file having candidate migration lines. We do not find any code changes in the remaining migrations.

3.3.2 Iterative Labeling Process. We find that the 1,269 migrations in SALM span 25 library domains, as labeled by the original SALM authors. Thus, when sampling the subset of migrations to manually analyze, we keep the following goals in mind: (1) the selected migrations are diverse across the

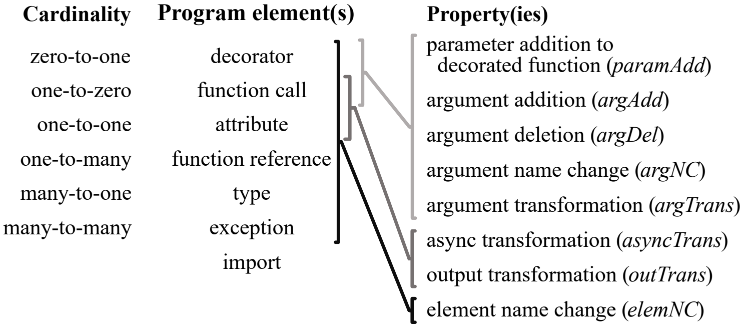

Cardinality
zero-to-one
one-to-zero
one-to-one
one-to-many
many-to-one
many-to-many
Program element(s)
decorator
function call
attribute
function reference
type
exception
import
Property(ies)
parameter addition to decorated function (*paramAdd*)
argument addition (*argAdd*)
argument deletion (*argDel*)
argument name change (*argNC*)
argument transformation (*argTrans*)
async transformation (*asyncTrans*)
output transformation (*outTrans*)
element name change (*elemNC*)

Fig. 3. PyMigTax: a taxonomy of Python migration-related code changes

different library domains to avoid biasing the types of code changes we observe to certain domains, (2) the sample size is statistically representative, and (3) the final selection is random.

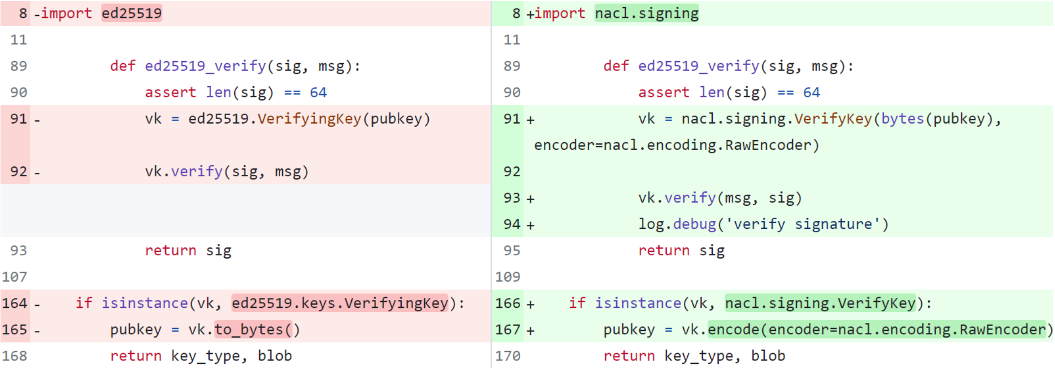
To satisfy these criteria, we perform a proportional stratified random selection [10, 36, 48] as follows. For statistical representativeness, we need to label 296 out of the total 1,269 migrations (5% margin of error, 95% confidence level), or 23.33% of the migrations. To ensure both diversity and randomness of the dataset, we perform this selection based on the migration count in each domain, selecting 23.33% of migrations from each domain, and rounding up. However, to ensure adequate data from all domains, we select a minimum of two migrations from each, unless only one is available. Due to this approach, we finally select a total of 311 migrations. Our artifact depicts the distribution across application domains of the library pairs.

We follow an iterative closed-coding approach [50] to label code changes in sampled migrations using the initial version of PyMigTax. However, we also allow raters to identify new types of code changes. In each round, we randomly select a number of migrations for labeling. Each migration is independently labeled by two authors. Disagreements are discussed and resolved after each round, involving all authors if needed. This process continues until substantial inter-rater agreement is achieved, and no new taxonomy labels are identified. Inter-rater agreement is measured using Cohen's kappa [17] score for program elements and cardinality, aiming for a score of ≥ 0.8 for near-perfect agreement [39]. For properties, which can have multiple labels per code change, we use Krippendorff's alpha [37], maintaining a customary threshold of ≥ 0.8.

The middle section of Table 1 shows the labeled migrations per iteration and the corresponding kappa/alpha scores. In the first round, we observe three new program elements and two new properties. In the second round, we find three new properties but no new program elements. In the third round, after labelling a total of 52 migrations with 371 migration-related code changes, we do not observe any new taxonomy items and also reach the target agreement for program elements, cardinality, and properties. Subsequently, each of the remaining 259 migrations, comprising 1,233 related code changes, is labeled by a single author, with no new items added to the taxonomy. The bottom section of Table 1 summarizes the labeled data used to construct PyMigTax.

3.4 A description of PyMigTax

Figure 3 illustrates the final version of PyMigTax, showing the three dimensions of a code change: cardinality (left), source/target program elements (middle), and properties (right). These items can be combined in various ways to depict a given code change. The connecting lines on the right show the observed properties for specific program elements, further detailed below. The text in parentheses next to a property represents its short name, frequently used throughout the paper.



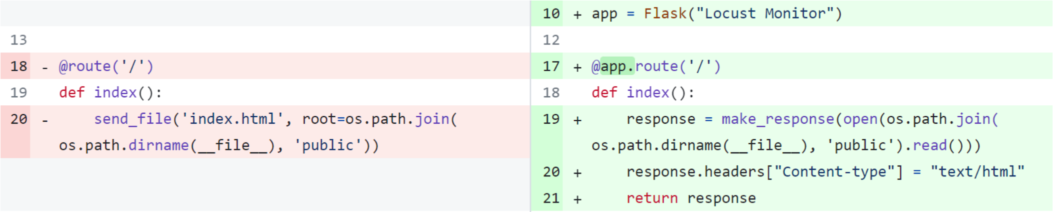
```
8 -import ed25519
11
89          def ed25519_verify(sig, msg):
90              assert len(sig) == 64
91 -            vk = ed25519.VerifyingKey(pubkey)

92 -            vk.verify(sig, msg)


93              return sig
107
164 -    if isinstance(vk, ed25519.keys.VerifyingKey):
165 -        pubkey = vk.to_bytes()
168          return key_type, blob
```

```
8 +import nacl.signing
11
89          def ed25519_verify(sig, msg):
90              assert len(sig) == 64
91 +            vk = nacl.signing.VerifyKey(bytes(pubkey),
 encoder=nacl.encoding.RawEncoder)
92
93 +            vk.verify(msg, sig)
94 +            log.debug('verify signature')
95              return sig
109
166 +    if isinstance(vk, nacl.signing.VerifyKey):
167 +        pubkey = vk.encode(encoder=nacl.encoding.RawEncoder)
170          return key_type, blob
```

Fig. 4. *ed25519* to *pynacl* migration extracted/adapted from *romanz/trezor-agent#e1bbdb4*.



```

13
18  - @route('/')
19    def index():
20  -     send_file('index.html', root=os.path.join(
    os.path.dirname(__file__), 'public'))
```

```
10  + app = Flask("Locust Monitor")
12
17  + @app.route('/')
18    def index():
19  +     response = make_response(open(os.path.join(
    os.path.dirname(__file__), 'public').read()))
20  +     response.headers["Content-type"] = "text/html"
21  +     return response
```

Fig. 5. *bottle* to *flask* migration extracted/adapted from *heyman/locust#4067b92*.

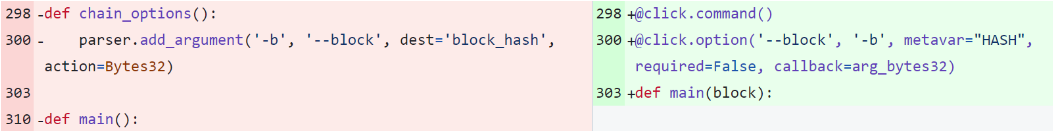
3.4.1 Program elements. Each code change involves specific program elements, with seven distinct types observed in our data. The first type, *function call*, includes *calls* to functions, methods, constructors, etc., commonly referred to as callable[1]. *Function reference*, on the other hand, occurs when a callable is only used as a reference, without calling it. In Figure 2, note how the two uses of the `netaddr.IPAddress` function in lines −56 and −419 are labeled as function call and function reference, respectively.

Similar to how PyMigTax distinguishes between function calls and function references, it also distinguishes between a *type* and a call to a constructor (the latter being a function call). In Figure 4, `VerifyingKey` (−164) is replaced with `VerifyKey` (+166), with both the source and target program elements being type. Libraries often use their own exception types, which also require migration. Such program elements are labeled as *exception* in PyMigTax.

An *attribute* indicates access to an attribute of an object, as illustrated by the removal of access to the attribute `value` in line −56 of Figure 2. A *decorator* denotes the use of a decorator, as shown in lines −18 and +17 of Figure 5. During migration, developers typically replace the import statements of the source library with those of the target library, a change described by the *import* program element in PyMigTax, depicted in lines 8:8 of Figure 4. Given the stylistic variations in importing a library or its APIs, import code changes are not labeled with cardinality or properties.

3.4.2 Cardinality. We find code changes having a total 6 different cardinalities, including one-to-one, one-to-many, many-to-one, and many-to-many, which are frequently mentioned in the literature [5, 54]. Cases also exist where APIs are either only removed or added, indicating one-to-zero or zero-to-one cardinalities, respectively. Figure 5 showcases a migration between the web

[1] https://docs.python.org/3/glossary.html#term-callable



```
298 -def chain_options():
300 -    parser.add_argument('-b', '--block', dest='block_hash',
     action=Bytes32)
303
310 -def main():
```

```
298 +@click.command()
300 +@click.option('--block', '-b', metavar="HASH",
     required=False, callback=arg_bytes32)
303 +def main(block):
```

Fig. 6. *argparse* to *click* migration extracted/adapted from *clearmatics/ion#03fb3a3*. .

application frameworks *bottle* and *flask*, illustrating a zero-to-one cardinality. Here, the source library, *bottle*, uses a static method (`route`) for routing, while the target library, *flask*, uses an instance method (`Flask.route`), requiring the initialization of the *Flask* object in line +10.

3.4.3 Properties. Each code change can additionally be characterized by properties. The property *element name change* applies when the names of the equivalent source and target APIs differ, such as the replacement of the function call `Bottle()` with the function call `Flask()` in Figure 1, which involves an element name change.

Code change 300:300 in Figure 6 exhibits several argument-related properties in the replacement of `add_argument()` with the decorator `@option()`. *Argument name change* apply to instances where the source and target APIs have identical arguments, but their names differ, as seen with `action` and `callback`. *Argument addition* occurs when one or more arguments are added to the target function call or decorator during migration, exemplified by `metavar` and `required`. Conversely, *argument deletion* involves the removal of an argument, such as `dest`.

Notice how the newly decorated function `main` in Figure 6 now takes a new parameter (`block`) to replace the original command line arguments with the same names in −300. PyMigTax has the property *parameter addition to decorated function* to describe this change. Note that parameter addition to decorated function differs from argument addition as the former affects a client code function, while the latter applies to the target API.

Sometimes, added and removed program elements take similar arguments, but argument changes are needed during migration. PyMigTax uses the *argument transformation* property to label such changes. Figure 4 illustrates this with `VerifyingKey` and `VerifyKey` from source and target libraries, both requiring a key, but the latter expects it in byte format, necessitating an additional `bytes` call on line +91. Another example involves the `verify` methods in both libraries, taking identical arguments but with reversed positions. Figure 5 depicts a third instance, where `send_file` and the corresponding `make_response` function in the target library have differing requirements.

PyMigTax features the *output transformation* property to address cases where source and target APIs return data in differing formats, thus requiring transformation during migration. In Figure 2, the source API provides a `value` attribute to access the returned IP address as an integer, whereas the target library recommends using the built-in `int` function for integer representation, exemplifying an output transformation property through the `int` call. The *async transformation* applies when either source or target APIs use async/await keywords, as shown lines 36:38 in Figure 7.

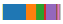
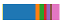
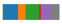
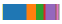
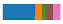
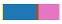
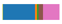
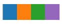
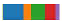
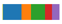
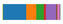
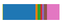
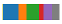
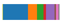
Note that some code changes may not have any properties. For example, the decorators replaced in 18:17 of Figure 5 have identical names and signatures, a code change with no properties.

3.5 PyMigBench-2.0

We used PyMigBench [31] and a subset of SALM [25] to build PyMigBench-2.0. Similar to PyMigBench [31], we store our labeled data in a YAML format, one YAML file for each migration. While the original PyMigBench included only the locations of each code change, PyMigBench-2.0 includes the program elements, cardinality, and the additional properties we label using PyMigTax. Additionally, a code change entry includes the names of the source and target APIs removed and

Table 2. Distribution of types of API mappings in PyMigBench-2.0

🟦 elemNC 🟧 argAdd 🟩 argDel 🟥 argNC 🟪 argTrans 🟫 asyncTrans ■ outTrans ■ paramAdd ■ no properties

		Target program elements								
		function call	**attribute**	**decorator**	**function reference**	**type**	**exception**	**import**	**none**	**total**
Source program elements	**function call**	103 (73%) 	16 (11%) 	3 (2%) 	-	-	-	-	24 (17%)	109 (77%) 
	attribute	9 (6%) 	18 (13%) 	-	-	-	-	-	3 (2%)	25 (18%) 
	decorator	1 (0.7%) 	-	8 (6%) 	-	-	-	-	2 (1%)	9 (6%) 
	function reference	-	-	-	3 (2%)	-	-	-	1 (0.7%)	4 (3%)
	type	-	-	-	-	10 (7%)	-	-	1 (0.7%)	10 (7%)
	exception	-	-	-	-	-	11 (8%)	-	4 (3%)	14 (10%)
	import	-	-	-	-	-	-	137 (97%)	9 (6%)	138 (98%)
	none	15 (11%)	5 (4%)	4 (3%)	1 (0.7%)	3 (2%)	3 (2%)	8 (6%)	-	23 (16%)
	total	107 (76%) 	30 (21%) 	12 (9%) 	4 (3%)	11 (8%)	12 (9%)	137 (97%)	31 (22%)	141 (100%) 

added, respectively, in this code change. The "Having code change" row in Table 1 shows the data included in PyMigBench-2.0. The dataset and its documentation are available in our artifact.

4 EMPIRICAL STUDY

In our empirical study, we answer three main research questions.

RQ1 What are the common types of API mappings that appear during migration between Python libraries?

- **RQ1.1** What program elements are typically involved in API mappings?
- **RQ1.2** What properties are typically involved in the code transformations for different types of API mappings?

RQ2 What combinations of code changes are common in Python library migrations?

RQ3 How much development effort is needed for Python library migrations?

4.1 RQ1 What are the common types of API mappings that appear during migration between Python libraries?

4.1.1 Motivation. Migrating from one library to another requires modifying the code from using the old to the new library. While some analogous libraries have similar APIs, requiring minimal adjustments, others may necessitate extensive modifications due to differing APIs. The data we labeled using PyMigTax helps us to understand typical *API mappings* from source to target libraries.

4.1.2 Methods. We answer this RQ by analyzing the distribution of program elements and properties that appear in the API mappings we infer from the code change in PyMigBench-2.0.

Consistent with existing literature [5, 7, 53], we derive API mappings from the code changes we observe. For example, the code changes 91:91 and 164:166 in Figure 4 both replace `VerifyingKey` with `VerifyKey`. Therefore, `VerifyingKey` → `VerifyKey` is an API mapping. An API mapping may appear in several code changes, or even several migrations. In that case, the corresponding code changes will all have the same source and target APIs; however, the properties involved in the code change may be different. The specific properties for each code change depend on the underlying client code and do not change the API mapping. Therefore, when studying API mappings in **RQ1**, we simply combine all observed properties for an API mapping. Suppose *foo()* → *bar()* API mapping requires element name change in one migration and argument addition and element name change in another migration, then we would have a single set of properties *{element name change, argument addition}* for this API mapping. Due to this difference in the nature of program elements and the corresponding differences in data analysis, we divide this RQ into two research questions for clarity.

In **RQ1.1**, given a library pair, we aim to identify the types of source and target program elements that appear in the API mappings for that library pair. Table 2 presents our findings for **RQ1.1**, as the top number and percentage in each cell. The columns and rows depict different types of program elements from PyMigTax. The number at the top of each cell shows the number of *library pairs* for which we find at least one API mapping between the respective program elements. The percentage is then calculated from the total of 141 unique library pairs in the dataset. For example, the first cell in row two indicates that 9 (6%) library pairs have at least one API mapping where an attribute is replaced with a function call. *None* indicates no source/target elements, representing API additions/removals, and a dash (-) denotes combinations of elements that we did not observe.

In **RQ1.2**, we explore the properties that occur during client code transformation. Specifically, we are interested to know out of all observed API mappings between two types of program elements (regardless of which library pair they belong to), how many API mappings required a specific property during the corresponding client code transformations. Thus, it is important to note that for properties, the unit of measurement is *not* a library pair, but rather an API mapping. We also use Table 2 to summarize the results, focusing on the stacked bar charts for properties. The table's colored legend shows different properties, and each cell displays the distribution of these properties for API mappings between the respective program elements. For example, the table reveals that *all* API mappings that required changing a type from the source library to a type in the target library only exhibited element name change.

4.1.3 *RQ1.1 Findings (Program Elements in library pairs).*

We now discuss the top number in each cell from Table 2 related to program elements. We observe that import replacements are found in almost all library pairs (97%), as libraries typically have distinct import names, prompting changes during migration. However, it is interesting to note that some libraries, aiming for easy migration, maintain identical import names. We find 2 such pairs in PyMigBench-2.0: *PyCryptoDome* and *PyCrypto*, both using the import name `Crypto`, and *py-bcrypt* and *bcrypt*, both using `bcrypt`.

Given that functions are the main type of API offered by libraries, it is not surprising to see that most library pairs (77%) have API mappings involving function calls as a source element, and 76% as a target element. Interestingly, function calls are not always replaced by other function calls; they can also be replaced by attributes (11% of library pairs) and, less frequently, decorators (2% of library pairs). The opposite is also true: attributes can be replaced with function calls (6%) and decorators replaced by function calls (0.7%).

Replacements involving function reference, type, and exception replacements are less common, found in only 3%, 7%, and 10% migrations respectively. Function reference and type are only replaced with the same type of program element or are removed. While most migrations replace source APIs with target APIs, we also observe some cases of addition (*none* column, 22% library pairs) and removal (*none* row, 16% library pairs).

4.1.4 *RQ1.2 Findings (Properties in API mappings).*

Turning to the properties in code transformations, the predominant blue in in the stacked bar charts of Table 2 highlights frequent element name change. It suggests that the APIs from the source library do not necessarily map to APIs with the same name in the target library. Nevertheless, 32% of function call to function call API mappings retained the same name in both libraries.

Argument-related properties only apply to function calls and decorators, as other program elements do not take arguments. We observe argument addition (orange) in 28% of function call to function call API mappings and 50% of function call to decorators API mappings. We also see that argument addition occurs in 100% of decorators to function call API mappings and 20% of decorators to decorators API mappings. We find that argument deletion (green) is less common than argument addition (total 18% vs 22%). Specifically, they are common in API mappings with a decorator as a target program element (67%). On the other hand, argument transformation (purple) is common in API mappings with function call as a source program element (28%). We observe that 30% of decorator to decorator API mappings involve argument name change; however, none of the decorator to function call API mappings have argument name change.

Async transformation and output transformation are only applicable to function calls and attributes, but do not occur frequently (4% and 7% respectively of all API mappings). We observe Parameter addition to decorated function even less frequently (2% of all API mappings). Recall that we do not label properties for import code changes. Also, there are no properties when an API is just added or removed. Therefore there are no charts in the import and none rows and columns.

RQ1: The majority of library pairs (73%) have API mappings between function calls. However, we also find that 40% of library pairs have API mappings that involve non-function program elements. Regardless of the type of program elements in an API mapping, 65% of API mappings involve program elements with *different* names, and 29% of function call to function call API mappings involve argument transformation.

4.1.5 ***RQ1*** *discussion.* To the best of our knowledge, most of the current API mapping or client code transformation techniques consider function calls as the main program element to find mappings for [5, 6, 8, 45, 53]. If we assume these techniques work for Python, then the good news is that this implies that they may be able to support some API mappings in 73% of the library pairs in our data that have function call to function call API mappings. However, most of these techniques do *not* consider other program element types (e.g., attributes or decorators), and worse, do not try to deduce mappings between different types of program elements [5, 6, 8, 45, 53]. Our findings show that 40% of library pairs require replacements that involve non-function program elements.

Most of the previous library migration work focuses on Java [5, 7, 53], which does not support the async/await style programming. Supporting the async transformation we found in Python migrations would require adaptations to these techniques.

4.2 RQ2 What combinations of code changes are common in Python library migrations?

4.2.1 Motivation. In **RQ1**, we analyze the program elements and properties involved in the API mappings we find. However, a migration may require various code changes related to different API mappings. To fully understand the type of tool support needed for observed migrations, we must also examine the combinations of code changes in a complete migration. For example, our findings in **RQ1** reveal that 73% of libraries have API mappings that include function call as both source and target. However, the associated migrations may also include code changes related to attributes. Thus, in this RQ, we look at *migrations* in terms of combinations of the code changes that occur.

4.2.2 Methods. For each migration, we initially identify the unique combinations of code changes in terms of their program elements and properties. For instance, the migration depicted in Figure 7 has four code changes: code changes 35:36-37 and 98:100-101 have *function call → function call | elemNC, asyncTrans, argAdd* and code changes 36:38 and 99:102 have *attribute → function call*

Table 3. Code change combinations in the migrations.

rank	combination	# migs	% migs
1	function call → function call | elemNC	21	8.2%
2	function call → function call | no properties	18	7.0%
3	function call → function call | argTrans, elemNC	11	4.3%
4	function call → function call | argAdd, argDel, elemNC	8	3.1%
5	function call → function call | elemNC, outTrans	4	1.6%
6	function call → function call | argDel, elemNC	4	1.6%
7	function call → function call | argAdd	4	1.6%
8	function call → function call | argAdd, elemNC	4	1.6%
9	function call → function call | argTrans	4	1.6%
10	decorator → decorator | argTrans	3	1.2%
11	function call → function call | elemNC function call → function call | no properties	3	1.2%
12	function call → function call | argDel	3	1.2%
	remaining 155 combinations	169	66%
	total (167 combinations)	**256**	**100%**

| *elemNC, asyncTrans* Therefore, the distinct combination of code changes in this example is (1) *function call → function call* | *elemNC, asyncTrans, argAdd* and (2) *attribute → function call* | *elemNC, asyncTrans* . Next, for each unique combination of code changes, we determine the number of migrations that exhibit this combination. Through this analysis, we identify a total of 167 unique combinations (the full list is included in our artifacts).

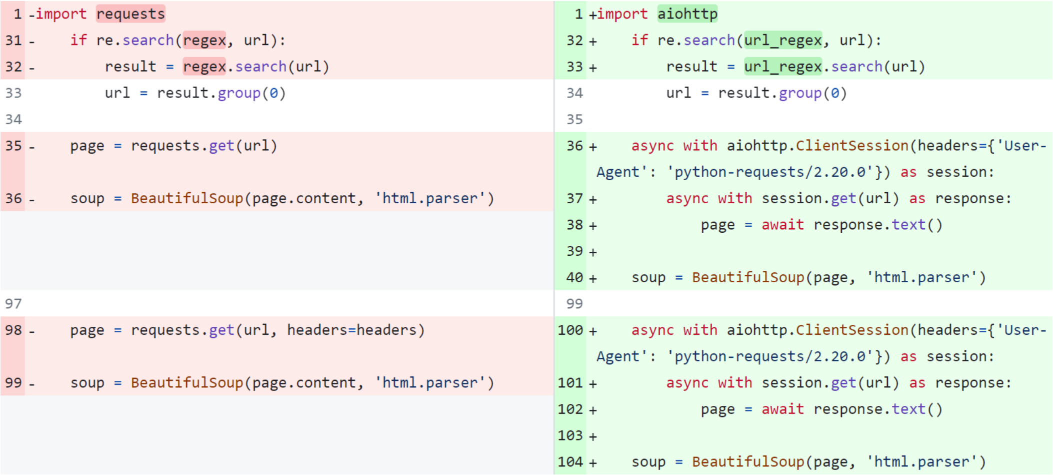
Table 3 displays the 12 most frequent combinations with at least 3 migrations, which we discuss in the paper. We note how row 11 shows a combination that has two different types of code changes, due to the properties. Since an import statement change is usually required for all migrations, we ignored import code changes when determining the combinations. This is why the Table 3 shows a total of 256 migrations instead of 335 as indicated in Table 1. For the remaining 79 migrations, the only changes that occurred in the migration were changes to the import statements.

4.2.3 Findings. From Table 3, the most frequent combination is function call replacements with element name change (8.2%) followed by those with no properties (7%). Within the function call replacements, we find several repeated property combinations, such as row 3 "argTrans, elemNC" (4.3%), row 4 "argAdd, argDel, elemNC" (3.1%), and 5 others. Overall, we observe 62 different combinations of properties associated with migrations that involve function call replacement, which suggests that performing the client code transformation requires intricate client code changes.

While 152 of the 167 unique combinations (91%) involve function calls in some way, 37% of the migrations combine functions with other program elements, whether in the same code change or in a different code change. In 9% of all migrations, we observe attributes combined with function calls within the same code change, while in 13% migrations, function call and attribute appear in the same migration but separate code changes. Similarly, decorators pair with function call in the same code change in 8% of migrations, whereas 11% of migrations have decorators in a different code change. While exception, type, and function reference appear with function calls in 5%, 3.5%, and 1.6% migrations, respectively, they do not appear in the same code change. Overall, 56% migrations involve only function calls, 7% migrations involve no function calls, and the remaining 37% involve function calls along with other program elements.

RQ2: 44% migrations are non-trivial, involving code changes with more than one type of program element and a variety of properties. Overall, only 16.4% migrations involve just function calls without any argument or output modifications.

4.2.4 ***RQ2*** *discussion.* Most existing library migration techniques focus on function calls as the APIs to migrate [5, 8, 45, 53]. Conceptually, these would not be able to fully support 44% of the PyMigBench-2.0 migrations with non-function program elements. Numerous techniques identify



```
 1 -import requests
31 -    if re.search(regex, url):
32 -        result = regex.search(url)
33          url = result.group(0)
34
35 -    page = requests.get(url)

36 -    soup = BeautifulSoup(page.content, 'html.parser')



97
98 -    page = requests.get(url, headers=headers)

99 -    soup = BeautifulSoup(page.content, 'html.parser')
```

```
  1 +import aiohttp
 32 +    if re.search(url_regex, url):
 33 +        result = url_regex.search(url)
 34          url = result.group(0)
 35
 36 +    async with aiohttp.ClientSession(headers={'User-
   Agent': 'python-requests/2.20.0'}) as session:
 37 +        async with session.get(url) as response:
 38 +            page = await response.text()
 39 +
 40 +    soup = BeautifulSoup(page, 'html.parser')
 99
100 +    async with aiohttp.ClientSession(headers={'User-
   Agent': 'python-requests/2.20.0'}) as session:
101 +        async with session.get(url) as response:
102 +            page = await response.text()
103 +
104 +    soup = BeautifulSoup(page, 'html.parser')
```

$MigLOC = 10, NumAPIs = 10, NumChanges = 4, UniqueAPIs = 5, UniqueMappings = 2$

Fig. 7. *requests* to *aiohttp* migration extracted/adapted from *raptor123471/dingolingo#1d8923a*.

only function API mappings, neglecting modifications in arguments or outputs [5, 8, 53]. Only 16.4% of observed migrations had simple function call replacements without additional properties.

4.3 RQ3. How much development effort is needed for Python library migrations?

4.3.1 Motivation. While RQ1 and RQ2 offer insights into the typical code changes required for library migration, they do not provide information on the associated development effort. In this RQ, we analyze our data to estimate the development effort needed for library migration, whether undertaken by a developer or an automated tool.

4.3.2 Methods. Although various metrics exist in software engineering [21], to the best of our knowledge, none directly measure the development effort required for a migration. Drawing inspiration from metrics used to characterize *code changes* in general [12], we define five simple metrics to estimate migration development effort. Specifically, our collected migration data allows us to measure the development effort from two perspectives: the migration size and the minimum amount of APIs developers need to learn for a migration.

M1 *MigLOC* is the sum of added and removed migration-related lines in non-import code changes in a migration. It is an estimation of the size of a migration. The diff in Figure 7, has 7 removed lines (−1, 31-32, 35-36, 98-99) and 13 added lines (+1, 32-33, 36-40, 100-104). Lines −31-32, +32-33, +39-40 and +103-104 are not migration related and lines −1 and +1 are import changes. Therefore, there are 4 removed and 6 added migration-related added lines, making MigLOC=10.

M2 *NumAPIs* is the total number of source and target API usages that were removed and added, respectively, in a migration. This provides an indication of the development effort required to find source API usages and replace them with corresponding target APIs. In Figure 7, the hunk 35-36:36-40 uses two source APIs `get()` and `content` and three target APIs `ClientSession()`, `get()`, and `text()`, totalling to 5 API usages. The same API uses are repeated in hunk 98-99:100-104, making NumAPIs for this diff = 5+5 =10.

M3 *NumChanges* is the total number of code changes in a migration. To replace source API usages, developers also need to understand the context (client code) in which the API has

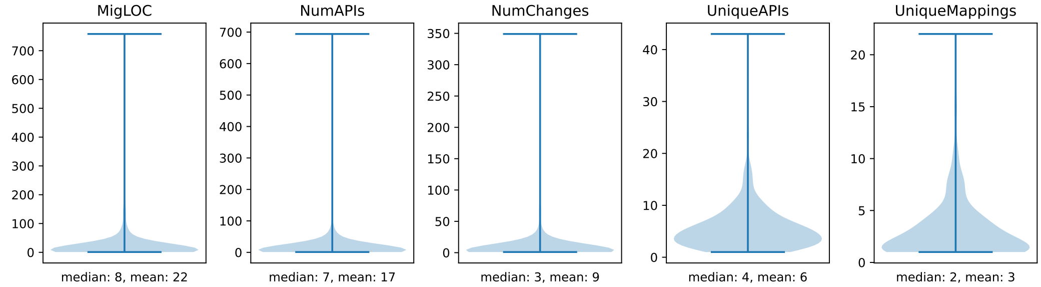


Fig. 8. Distribution of estimations of development effort.

been used, which takes effort. Code changes in a migration can be intermittent, which means developers need to understand different contexts. NumChanges provides an indication of the effort required to understand the context. In the example in Figure 7, there are a total of 4 non-import code changes (35:36-37, 36:40, 98:100-101 and 99:102). , making NumChanges=4 for this diff.

M4 *UniqueAPIs* is the number of *unique* APIs from the source and the target library used in a migration. When applying a migration, the developer has to understand the source and target APIs involved in the migration. Accordingly, *UniqueAPIs* is a proxy for the learning effort required for the migration. In Figure 7, hunk 35-36:36-40 uses 2 source and 3 target APIs, which are repeated in hunk 98-99:100-104, making UniqueAPIs=5 for this migration.

M5 *UniqueMappings* Other than the source and target APIs themselves, developers also have to learn which source API(s) should be replaced by which target API(s), i.e., API mapping. We use *UniqueMappings* to count the number of unique API mappings in a migration. The migration in Figure 7 has a total of 4 non-import code changes: 35:36-37, 36:38, 98:100-101 and 99:102. However, both 35:36-37 and 98:100-101 replace the same function calls. Similarly, code changes 36:38 and 99:102 are also identical. Therefore, UniqueMappings=2 in this migration.

4.3.3 ***RQ3*** *Findings.* Figure 8 shows the distribution of the five metrics. We use the medians of these distributions to describe a typical migration. On average, a migration involves 8 changed lines of code, 7 API instances, and 3 code changes, representing the development effort. From a learning standpoint, a developer typically needs to learn 4 APIs and 2 API mappings per migration. We find that for all metrics, the mean values are much higher than the corresponding median value. This means that the distributions are positively skewed, i.e., there are some migrations requiring very high development effort. Recall that our dataset includes only single-commit migrations, suggesting the displayed values already represent a lower bound.

Some migrations may demand high development effort but not necessarily high learning effort. For example, the migration from library *umsgpack* to *msgpack* in commit logicaldash/lise#028d0b34 involves substantial code changes (MigLOC=116, NumAPIs=72, NumChanges=36) but only necessitates learning 6 unique APIs and 3 mappings, indicating many repeated changes for the same mapping. Conversely, other migrations, like from library *twitter* to *tweepy* in commit cloudbotirc/-cloudbot#f8243223, necessitate both high development and learning effort due to larger differences between the libraries, requiring learning 35 unique APIs and 17 mappings.

RQ3: A typical Python library migration requires changing 8 LOC and 7 API instances, as well as understanding 4 APIs and 2 API mappings. However, some library migrations require higher number of lines of code and API changes as well as API understanding and mapping, reaching up to changing 758 lines of code, understanding 43 APIs, and mapping 22 APIs.

4.3.4 ***RQ3*** *Discussion.* The good news for both developers and tools is that the average migration does not require significant development and learning effort. However, some migrations do require considerable effort, highlighting the need for tool support to alleviate the manual burden of migrating libraries. Existing API mapping tools and techniques [15, 45] can reduce learning effort. However, as pointed above, comprehensive client code transformation tools are still missing. That said, not all migrations with high development effort necessarily require high learning effort, it could be that there are many repeated code transformations in a migration. However, such changes are still not simple "find and replace" or a well-defined refactoring that a developer can apply on a whole file. One idea could be to develop tools that observe the developer as they make one of these changes and infer the change pattern that could be applied to the rest of the file (e.g., something similar to learning from examples [35]). In fact, it would be interesting to see if Large Language Models already integrated in the IDE, e.g. Copilot, might be effective at such a task.

5 IMPLICATIONS

The contributions of this paper as well as the findings from our empirical study have several implications for library migration researchers and tool builders.

5.1 Implications for API mapping research

PyMigBench-2.0, unlike the original PyMigBench, includes APIs from each of the 3,096 manually verified code changes in the dataset. This serves as a ground truth, aiding researchers in evaluating and training their API mapping techniques. as well as train their techniques using this data. While most of the existing API mapping techniques focus on one-to-one API mapping, PyMigBench-2.0 includes code changes that involve higher cardinality API replacements. This indicates a need for higher cardinality API mapping techniques, which PyMigBench-2.0 can facilitate.

5.2 Implications for client code transformation research

As discussed in RQ2, we find that the client code transformations required in real migrations go beyond only identifying API mappings, and also beyond focusing on mappings between the same types of program elements. From RQ1, we also find that some migrations require taking synchronizations into account, especially if one library supports asynchronous programming while the other does not. Migration tools that can automatically infer the need to add `async`/`await` keywords are needed. Overall, our characterization of migration-related code changes and analysis of their frequency in typical migrations allow tool developers to make informed decisions about the type of client code transformation they should support. Additionally, the labeled and validated code changes we provide in PyMigBench-2.0 also facilitate the evaluation of any developed migration techniques.

5.3 Implications for systematic labeling and comparisons

Researchers can use PyMigTax to label the capabilities of their techniques and fairly compare library migration techniques and identify their limitations. For example, as a conceptual labeling, SOAR [45] supports one-to-one and one-to-many function call replacements along with function name change and argument addition, deletion, and transformation. However, SOAR does not support decorators, attributes, types and many-to-one or many-to-many function call replacements present in PyMigTax. Moreover, SOAR does not support async transformation, which means it cannot migrate between synchronous and asynchronous libraries. Together, PyMigTax and PyMigBench-2.0 allow both conceptual and empirical comparisons of library migration techniques. This allows the identification of limitations of existing techniques and then developing new techniques to address these limitations.

6 THREATS TO VALIDITY

6.1 Internal validity

To enhance the accuracy and correctness of SALM and reduce manual effort in identifying code changes, we performed some automated steps. We used OpenAI's GPT-4 API to identify non-analogous libraries and to extract libraries' import names. The results produced by GPT-4 may not be completely reliable. We manually verified 25 (5% of all) of GPT-4's results and 21 of 25 (84%) were correct. GPT-4 marked the remaining four as non-analogous, but we found them to be analogous. We acknowledge that if GPT-4 incorrectly indicates that two libraries are non-analogous, we will miss the related migrations completely. That said, our chosen methodology matches our goal of ensuring that migrations that end up in our benchmark are actually migrations.

We also ignored the libraries that are not available in PyPI and their corresponding migrations, because we focus on only third-party libraries. In all these steps, we may have filtered out true migrations between analogous third-party libraries and thus missed their code changes. To minimize this, we validated import names and ignored migrations without code changes during manual review. Since our manually verified migrations from SALM cover 100% of the library domains and 66% of the library pairs from the original dataset, it is unlikely that any overlooked migrations would significantly differ from our observations. This assumption is supported by reaching data saturation after labeling 52 migrations from SALM.

In identifying code changes, we considered only visible modifications detected by git's diff, thereby missing instances where an API is replaced with an identical one (e.g., the same qualified name and signature for function calls). While this could have potentially increased the percentage of program elements with no element name change, we deem such cases as less relevant for library migration research, given that identical APIs require no migration action.

In a migration, a source library could be replaced by many target libraries and vice versa. In this study, we assume that one library is replaced by one one target library. Although this does not affect our taxonomy, considering multiple source or target libraries may have increased the estimated development effort. Thus, our results are primarily applicable to one-to-one library migrations.

6.2 Construct validity

We manually review migrations to identify and label code changes, relying on the authors' library knowledge, which could potentially lead to mislabeling. To mitigate this, we reviewed each library's documentation and examples, ensuring sufficient understanding. We erred on the side of caution and skipped unclear migrations (e.g., commits with too many tangled changes and refactorings), clearly marking them as such. We also refined our results via several iterations of discussions among the authors and had two authors independently review each migration until achieving substantial inter-rater agreement and data saturation.

The decision to categorize a set of APIs involved in a migration as multiple one-to-one code changes or one higher-cardinality code change can often be subjective. We use the API usages, names, arguments, and documentation to determine if one-to-one relations can be established. In cases of ambiguity, we consistently categorize these as higher cardinality changes.

Our work focuses on migration-related changes rather than stylistic ones, so we do not consider changes in the qualified name as code changes during labeling; these can be inferred from the function name and import statement. Style-related changes were ignored in the identification and labeling process. All labeled migration data is available for external review and verification.

The metrics we use in RQ3 reflect various characteristic of a migration as *proxies* for development effort, and as inspired by existing software engineering and software evolution principles/metrics. We acknowledge these characteristics or metrics haven't been validated to directly correlate with perceived migration development effort. Additionally, we can only approximate the effort related to observable code changes, without information on testing or additional discussions between team members. Our aim is to provide some *perspective* of migration effort, with validating and improving these metrics being an interesting avenue for future work.

6.3 External validity

The datasets we use only consider single-commit migrations, but previous studies have shown that library migration can span multiple commits [53]. For this first effort on systematically characterizing Python library migrations, we limited ourselves to existing datasets. In the future, we hope to build multi-commit Python migration datasets and investigate whether multi-commit migrations are more complex than single-commit migrations.

PyMigTax is based on code changes between 141 library pairs containing 208 unique libraries across 35 domains. While we aimed for data saturation using two datasets and followed a stratified sampling approach to analyze a statistically representative sample of migrations, we cannot guarantee that we observed all types of code changes. We document our extraction and review process to allow others to extend PyMigTax and PyMigBench-2.0.

7 RELATED WORK

7.1 Library migration datasets and benchmarks

Teyton et al. [52, 54] and He et al. [28, 29] provide benchmarks for Java library migration. Teyton et al.'s initial work [52] developed a semi-automatic approach to detect 80 analogous Java library pairs by mining Maven repository histories and manually verifying the results. They later refined their methodology [54] to capture multi-commit migrations, ultimately identifying 329 library pairs across 32 domains. He et al. [29] took a different approach by using both Teyton et al.'s mining technique and their own recommendation system to validate a dataset comprising 1,434 analogous library pairs. Later, they expanded this to include 3,163 manually verified migration commits [28]. Both efforts share a common approach in terms of mining client repository history, filtering candidates based on specific metrics (e.g., selecting repositories with more than 10 stars, or analyzing the project dependency files), and manual verification. These collectively provide a robust methodological foundation for mining library migrations. These methods were also followed by the Python library migration datasets we use in this paper, SALM [25] and PyMigBench [31], and which we discussed in detail in Section 2.2. Our work relies on SALM and PyMigBench, but further expands these datasets by identifying, labeling, and categorizing migration-related code changes. Notably, aside from line numbers in PyMigBench, neither of the original datasets contains labeled code changes.

7.2 API mapping

API mapping techniques for library migration can be grouped into history-based and non-history-based approaches. *History-based methods*, represented by Teyton et al. [53] and Alrubaye et al. [5], analyze migration commits to discern and filter function mappings, addressing issues like higher cardinality mappings [8] and incorporating machine learning from API documentation [6]. In contrast, *non-history-based techniques*, like the work by Chen et al. [15] and SOAR [45], employ unsupervised deep learning models or textual similarity to identify API mappings. Non-history based techniques are especially useful when there are insufficient historical migration examples. Despite their merits, both techniques have limitations: history-based approaches struggle with new libraries [45], SOAR, requiring ample documentation, has been tested on only two domain-specific libraries (R and Python) [3, 45].

7.3 Client code transformation

The ultimate aim of library migration research is to develop tools that can automatically transform client code that uses one library to use a different library. Balaban et al. [9] developed a technique to migrate the uses of legacy Java classes. SOAR above [45] used program synthesis techniques for client code transformation based on identified API mappings, though it is been evaluated on just one Python library pair. SOAR is notably the only known technique for Python. Additionally, there is a substantial body of work on version migration and updating client code due to breaking changes, such as [14, 46, 57], addressing a related issue. While our study does not propose automated techniques, our labeling of migration-related code changes helps identify potential API mappings between analogous Python libraries. This lays foundational knowledge for future API mapping and code transformation methods, offering insights into the types of code changes to be addressed and supplying labeled benchmark data for evaluation.

7.4 Library migration taxonomies

To our knowledge, no standard taxonomy exists for types of migration-related code changes. While some techniques presented above [5, 6, 8, 45, 53] address various code changes, there is no systematic analysis of possible change types, making their selection unclear. For instance, Alrubaye et al. categorize API mappings into one-to-one, one-to-many, and many-to-many but only discuss functions. SOAR [45] considers the need for argument addition, deletion, and transformation but but is limited by the target library/domain. Our systematic approach to understanding Python code changes revealed previously unconsidered changes, such as replacements across different types of program elements).

7.5 Measuring migration complexity and effort

Defining reliable metrics in software engineering is a long-time challenge, with numerous proposed metrics to measure code complexity [2, 18, 21, 26, 27, 42], or qualitative levels of complexity [13]. There has also been research been devoted to estimating development effort [4, 23, 30, 32, 41, 51]. While some methods, like [4], use code-based metrics, most analyze software process, relying on requirements specifications, use cases, work breakdown structure, and other artifacts, often combined with expert knowledge. Our characterization of migration development effort through size related metrics and proxies for learning effort were inspired by the above literature as well as code change measurements [12].

8 CONCLUSION

We conducted an empirical study to gain a comprehensive understanding of Python library migrations. To enable this empirical study, we contributed PyMigBench-2.0, a manually curated library migration dataset, and PyMigTax, a taxonomy for categorizing migration-related code changes. PyMigBench-2.0 comprises a collection of 335 migrations sourced from two state-of-the-art migration datasets, which we have significantly enhanced by incorporating a 3,096 manually verified and labeled migration-related code change instances.

From our empirical study, we find that 40% of library pairs have API mappings that involve non-function program elements. We find that 16.4% of the migrations are trivial, involving only function calls replacements without any argument or output modifications. However, a larger proportion (44%) are non-trivial, involving code changes with more than one type of program element and a variety of properties. As an approximation for the development effort involved, we find that, on average, a developer needs to learn about 4 APIs and 2 API mappings to perform a migration, and change 8 lines of code. However, we also found cases of migrations that involve up to 43 unique APIs, 22 API mappings, and 758 lines of code, making them harder to manually implement. Overall, our contributions provide the necessary knowledge and foundations for developing and evaluating automated Python library migration techniques.

9 DATA AVAILABILITY

Replication package available at https://doi.org/10.6084/m9.figshare.24216858.v2.

ACKNOWLEDGEMENT

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) through their CREATE, Discovery, and Canada Research Chairs programs.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 385–395.

[2] Alain Abran. 2010. *Software metrics and software metrology*. John Wiley & Sons.

[3] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. 2020. Software documentation: the practitioners' perspective. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 590–601.

[4] Allan J. Albrecht and John E Gaffney. 1983. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE transactions on software engineering* 6 (1983), 639–648.

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

[6] 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.

[7] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 414–417.

[8] 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.

[9] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring support for class library migration. *ACM SIGPLAN Notices* 40, 10 (2005), 265–279.

[10] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 94.

[11] Veronika Bauer and Lars Heinemann. 2012. Understanding API usage to support informed decision making in software maintenance. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 435–440.

[12] Hans Christian Benestad, Bente Anda, and Erik Arisholm. 2009. Understanding software maintenance and evolution by analyzing individual changes: a literature review. *Journal of Software Maintenance and Evolution: Research and Practice* 21, 6 (2009), 349–378.

[13] B Boehm and D Reifer. 2000. Software Cost Estimation with COCOMO II. Prentice Hall. *Upper Saddle River, NJ* (2000).

[14] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 507–511. https://doi.org/10.1109/SANER.2018.8330249

[15] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2019. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*

47, 3 (2019), 432–447.

[16] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2021. Mining Likely Analogical APIs Across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering* 47, 3 (2021), 432–447. https://doi.org/10.1109/TSE.2019.2896123

[17] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[18] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, MA Borst, and Tom Love. 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering* 2 (1979), 96–104.

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

[20] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. 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*. 2187–2200.

[21] Norman Fenton and James Bieman. 2014. *Software metrics: a rigorous and practical approach*. CRC press.

[22] Python Software Foundation. [n. d.]. *Python Package Index - PyPI*. Retrieved March 31, 2022 from https://pypi.org

[23] Swarnima Singh Gautam and Vrijendra Singh. 2018. The state-of-the-art in software development effort estimation. *Journal of Software: Evolution and Process* 30, 12 (2018), e1983.

[24] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 12–21. https://doi.org/10.1109/MSR.2012.6224294

[25] 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)*. IEEE, 627–638.

[26] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[27] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software complexity analysis using halstead metrics. In *2017 international conference on trends in electronics and informatics (ICEI)*. IEEE, 1109–1113.

[28] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 478–490.

[29] Hao He, Yulin Xu, Yixiao Ma, Yifei Xu, Guangtai Liang, and Minghui Zhou. 2021. A multi-metric ranking approach for library migration recommendations. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 72–83.

[30] Peter R Hill. 2011. *Practical software project estimation: a toolkit for estimating software development effort & duration*. McGraw-Hill Education.

[31] 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)*. IEEE, 511–515. https://doi.org/10.1109/MSR59073.2023.00075

[32] Magne Jørgensen. 2014. What we do and don't know about software development effort estimation. *IEEE software* 31, 2 (2014), 37–40.

[33] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E Hassan. 2016. Logging library migrations: A case study for the apache software foundation projects. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 154–164.

[34] Jeremy Katz. 2020. Libraries.io Open Source Repository and Dependency Metadata. https://doi.org/10.5281/ZENODO.808272

[35] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and applying type changes. In *Proceedings of the 44th International Conference on Software Engineering*. 1206–1218.

[36] Barbara Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of Survey Research: Part 5: Populations and Samples. *SIGSOFT Softw. Eng. Notes* 27, 5 (sep 2002), 17–20. https://doi.org/10.1145/571681.571686

[37] Klaus Krippendorff. 2013. *Content analysis: An introduction to its methodology* (3rd ed.). Sage publications, Thousand Oaks, California. 221–250 pages.

[38] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

[39] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[40] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 245–256.

[41] Yasir Mahmood, Nazri Kama, and Azri Azmi. 2020. A systematic review of studies on use case points and expert-based estimation of software development effort. *Journal of Software: Evolution and Process* 32, 7 (2020), e2245.

[42] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[43] Israel J Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E Hassan. 2013. A large-scale empirical study on software reuse in mobile apps. *IEEE software* 31, 2 (2013), 78–86.

[44] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API Elements for Code Migration with Vector Representations. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 756–758. https://doi.org/10.1145/2889160.2892661

[45] Ansong Ni, Daniel Ramos, Aidan ZH Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2021. Soar: a synthesis approach for data science api refactoring. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 112–124.

[46] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Semantic patches for adaptation of javascript programs to evolving libraries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 74–85.

[47] R OpenAI. 2023. GPT-4 technical report. *arXiv* (2023), 2303–08774.

[48] Qualtrics. [n. d.]. How to use stratified random sampling in 2023. https://www.qualtrics.com/experience-management/research/stratified-random-sampling/.

[49] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2019. To react, or not to react: Patterns of reaction to API deprecation. *Empirical Software Engineering* 24, 6 (2019), 3824–3870.

[50] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.

[51] Ashish Sharma and Dharmender Singh Kushwaha. 2012. Estimation of software development effort from requirements based complexity. *Procedia Technology* 4 (2012), 716–722.

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

[53] 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.

[54] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A study of library migrations in java. *Journal of Software: Evolution and Process* 26, 11 (2014), 1030–1052.

[55] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. 2020. Exploring how deprecated python library apis are (not) handled. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 233–244.

[56] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.

[57] Zhenchang Xing and Eleni Stroulia. 2007. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (2007), 818–836.

[58] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. 2020. Why reinventing the wheels? An empirical study on library reuse and re-implementation. *Empirical Software Engineering* 25, 1 (2020), 755–789.

[59] 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*. 106–116.