Characterization and Automatic Updates of Deprecated Machine-Learning API Usages

Stefanus A. Haryono*, Ferdian Thung*, David Lo*, Julia Lawall†, and Lingxiao Jiang*
*School of Information Systems, Singapore Management University, Singapore
†Inria, France
{stefanusah, ferdianthung, davidlo, lxjiang}@smu.edu.sg
Julia.Lawall@inria.fr

***Abstract*—Due to the rise of AI applications, machine learning (ML) libraries, often written in Python, have become far more accessible. ML libraries tend to be updated periodically, which may deprecate existing APIs, making it necessary for application developers to update their usages. In this paper, we build a tool to automate deprecated API usage updates. We first present an empirical study to better understand how updates of deprecated ML API usages in Python can be done. The study involves a dataset of 112 deprecated APIs from Scikit-Learn, TensorFlow, and PyTorch. Guided by the findings of our empirical study, we propose MLCatchUp, a tool to automate the updates of Python deprecated API usages, that automatically infers the API migration transformation through comparison of the deprecated and updated API signatures. These transformations are expressed in a Domain Specific Language (DSL). We evaluate MLCatchUp using a dataset containing 267 files with 551 API usages that we collected from public GitHub repositories. In our dataset, MLCatchUp can detect deprecated API usages with perfect accuracy, and update them correctly for 80.6% of the cases. We further improve the accuracy of MLCatchUp in performing updates by adding a feature that allows it to accept an additional user input that specifies the transformation constraints in the DSL for context-dependent API migration. Using this addition, MLCatchUp can make correct updates for 90.7% of the cases.**

***Index Terms*—Python, Program Transformation, Automatic Update, Deprecated API**

I. Introduction

The popularity of machine learning (ML) has surged in recent years. Currently, the most popular programming language for ML is Python, due to the many Python libraries that support and provide ML capabilities via a set of Application Programming Interface methods (simply referred to as APIs). Applications (a.k.a. client code of the libraries) can then access the library functionality via these APIs. Libraries are typically updated to add new features, fix bugs, or improve performance. With each new update, changes in the library's API are inevitable, which may include API deprecation. API deprecation typically affects API methods, rendering them obsolete and requiring client code to be modified to invoke the updated API methods to ensure that the code works with the new version of the library (hereafter, we refer to a deprecated API method as a deprecated API).

Using the latest version of the ML library is important for the performance and security of the library client [1], [2]. Yet, updating usages of deprecated APIs can be cumbersome, causing many developers to stick to an older version despite the performance issue and security risk [1]–[3]. Hence, providing a better alternative to manually updating the usages of Python deprecated ML library APIs can benefit many developers.

In this work, we first conduct an empirical study to ascertain the level of innovation needed in the automatic update of ML libraries deprecated API usages and then design a solution to address the problem. We conduct our empirical study on three popular Python ML libraries, namely Scikit-Learn, TensorFlow, and PyTorch, by analyzing their API deprecations. Through this empirical study, we aim to discover deprecated API migration characteristics of popular Python ML libraries in multiple dimensions, i.e., the update operation in the API migration, the API migration mapping, and the API migration context dependency. For each dimension, we define some categories of deprecated API migrations. While the result of our study may also be applicable to other non-ML Python libraries, we purposely focus on the ML domain due to the rise of ML systems [4], [5] and ML libraries popularity.

Based on the findings in the empirical study, we propose a tool to automate the update of Python deprecated API usages called MLCatchUp. MLCatchUp takes as input the deprecated API signature, the updated API signature, and the file to be updated. It then automatically infers the required transformation by comparing the deprecated and updated APIs' signatures. The inferred transformations are expressed in the form of a Domain Specific Language (DSL) that can be easily understandable by MLCatchUp's users. We design this DSL based on the results of our empirical study.

Prior to this study, several attempts have been made to automate the update of deprecated API usages in other programming languages [6]–[9]. However, prior works primarily target Java programs, and none has targeted API deprecation in Python. Python has several unique features as compared to Java: it is a dynamically-typed language, it has two types of parameters (positional and keyword parameters), it offers parameter default values, etc. In particular, being a dynamically-typed language poses the following problems: (1) Variable types may change during code execution; (2) Information regarding variable types and function return types are typically unavailable. These problems render analysis and

```
1  def tensorize_file(file_reader):
2      data = []
3      for row in file_reader:
4          data.append(row)
5      data = torch.tensor(data)
6      mask = torch.tensor([[1., -1.], [1., -1.]])
7      return data.mul(mask)
```

Fig. 1: Example of variable type change in Python

transformation of Python code harder, as the type information is more limited compared to statically-typed language.

Consider the Python code snippet shown in Figure 1. In this code snippet, the variable `data` is first initialized as a `list` in line 2. In line 5, this `data` variable is used as a parameter to the `torch.tensor` API, which returns a `torch.Tensor` object. This `torch.Tensor` object is assigned into the `data` variable, changing the variable type. This `data` variable is then used to invoke the `torch.Tensor.mul` in line 7. If a tool cannot correctly determine that the type of the `data` variable in line 7 is `torch.Tensor` object, it would also not be able to determine that the `data.mul` method call used in line 7 is a `torch.Tensor.mul` API invocation. To address this problem, MLCatchUp is built with type inference functionality, leveraging state-of-the-art static analysis and deep learning tools [10], [11].

We have evaluated MLCatchUp on a dataset of 267 files containing 551 deprecated API usages involving 68 Python ML library APIs. Based on this evaluation, MLCatchUp can detect 93.7% of deprecated API usages without type inference. With the addition of type inference, it can detect deprecated API usages in all the cases, while it can perform correct updates 80.6% of all cases (*accuracy* of 80.6%). We have also conducted a qualitative study on the failed update results, where we found that one of the limitations of MLCatchUp is its inability to infer transformation constraints for context-dependent migrations, i.e., API migrations that depend on the value and type of the arguments used. To address this problem, we add a feature into MLCatchUp to accept such transformation constraints as an additional user input. With this feature, MLCatchUp achieved 90.7% accuracy.

In summary, the main contributions of this work are:

- We perform an empirical study on deprecated API usage migrations for popular Python ML libraries (Scikit-Learn, TensorFlow, and PyTorch), and characterize them in several dimensions.
- We create MLCatchUp, an automated Python API usage update tool with type inference that can infer the required API transformation automatically. To the best of our knowledge, this is the first such tool for Python. We design our tool to support the main migration characteristics of ML libraries that we identified in our empirical study.
- We evaluate MLCatchUp using a dataset of 551 deprecated API usages of popular ML libraries collected from public GitHub repositories, where MLCatchUp can achieve up to 90.7% accuracy.

The rest of this paper is organized as follows. Section II discusses our empirical study of Python deprecated API migrations and its findings. Section III presents the structure and features of our Python API usage automated update tool, MLCatchUp. Section IV describes our evaluation of MLCatchUp. Section VI discusses related work on API deprecation and program transformation. Section V discusses the threats to validity. Finally, Section VII concludes with future work.

II. Empirical Study

Through this study, we seek to get a better understanding of the characteristics of deprecated API migrations of Python ML libraries. Using findings of this study, we aim to create a tool to automate the update deprecated Python ML API usages. As highlighted in Section I, our goal is not to create a tool that supports all kinds of Python API migrations, but rather those that frequently appear for popular Python ML libraries.

A. Research Questions

Our empirical study analyzes the API migration characteristics of Python ML libraries, answering the following questions:

- **RQ1** *What are the characteristics of the API migrations for Python ML deprecated API usages?* We seek to identify *categories* (types) of API migrations, considering several *dimensions*. Understanding these characteristics will help us design an automatic deprecated API migration tool.
- **RQ2** *What is the distribution of deprecated API migration categories in Python ML libraries?* Understanding the frequency of each API migration category will help us to determine the most common API migrations, and what type of API migrations should be prioritized by an update tool.

B. Data Collection

For our study, we picked a subset of the available Python ML libraries based on the number of GitHub repositories that depend on the libraries (obtained through GitHub dependency graph[1]). As of August 2020, more than 120,000 repositories utilize Scikit-Learn, while more than 82,000 repositories use Tensorflow. Keras and Pytorch are used by at least 52,000 and 34,000 repositories, respectively. However, since Keras is included in TensorFlow since January 2017,[2] we picked Scikit-Learn, TensorFlow, and PyTorch as the subjects of our study.

Currently, there is no standard for deprecating ML library APIs. Thus, we collected the API deprecations by manually reading the change log of each library. We focused on looking for text or API methods that are marked with the word "deprecat-" and "replace-" in the change log, which indicates API deprecation or replacement. For Scikit-Learn, the change log is available in their official documentation page.[3] For PyTorch[4] and TensorFlow,[5] the change log can be found in their GitHub release note. We limit our manual collection to the change logs of major and minor release versions between

[1]https://docs.github.com/en/code-security/supply-chain-security/about-the-dependency-graph

[2]https://www.fast.ai/2017/01/03/keras/

[3]https://scikit-learn.org/stable/whats_new/v0.23.html

[4]https://github.com/pytorch/pytorch/releases

[5]https://github.com/TensorFlow/TensorFlow/releases

TABLE I: Number of Deprecated APIs in ML Libraries

Library	# Deprecated APIs
Scikit-Learn	22
TensorFlow	54
PyTorch	36
Total	112

July 2018 and August 2020. From the documentation, we found a total of 112 pairs of deprecated APIs and their updated APIs, distributed across the libraries as shown in Table I.

C. RQ1: Deprecated API Migration Characteristics

In this research question, we aim to identify different *categories* (types) of API migrations considering different dimensions. We utilized thematic analysis, which is a qualitative data analysis method for systematically identifying, organizing, and offering insight into patterns or themes across a dataset [12], [13]. In this process, we first familiarize ourselves with the data by thoroughly reading and analyzing the deprecated and updated signatures of the 112 deprecated APIs, as well as additional information from various documentation. We also investigate how the migrations from the deprecated to the updated APIs can be done. Next, in multiple iterations, we identify categories of migrations, across multiple dimensions.

Our thematic analysis identifies many categories (types) of API migrations that can be put in three dimensions. The first dimension (D1) is related to the *update operation* required to perform the migration. The second one (D2) relates to the *degree of relationship* (e.g., 1-to-1, etc.) between the deprecated and updated APIs. And the last one (D3) relates to the *context dependence* of the migration. The definitions of these dimensions and their categories are:

D1: Update Operation. We observe common patterns in the updates required to migrate a deprecated API to its updated APIs by looking at the differences between the signatures of the two APIs. We find the following categories:

- **Remove a parameter:** This operation removes the deprecated API usage function parameter(s). If the deprecated parameter is not removed, it will result in a TypeError, as the parameter no longer exists.
- **Rename a parameter:** This operation replaces one or more keyword parameter names with a new name. If the parameter name is not changed, it will result in an error.
- **Convert a positional parameter to a keyword parameter:** This operation removes one or more positional parameters and uses their values to create new keyword parameters. An example is shown in Figure 2. The second positional parameter of `torch.add` is deprecated and replaced with the keyword parameter `alpha`. The deprecated API usage is in line 4, while the updated code is in line 5.
- **Rename a method:** This operation renames the function or module of the deprecated API to a new updated name. If the module name is changed, an import statement that imports the updated API is also added.
- **Add a parameter:** This operation adds a new keyword/positional parameter to an API invocation.

```
1       import torch
2       t_0 = torch.randn((3, 3))
3       t_1 = torch.ones_like(t_0)
4   -   t_add = torch.add(t_0, 10, t_1)
5   +   t_add = torch.add(t_0, t_1, alpha=10)
```

Fig. 2: Changing a positional parameter to a keyword parameter to migrate a deprecated `torch.add` API usage from PyTorch version 1.5.0

```
1       import sklearn
2       def check_estimator(TestEstimator):
3   -       sklearn.utils.estimator_checks.check_
4   -               estimator(SklearnWrapperClassifier)
5   +       if isinstance(TestEstimator,
6   +               sklearn.base.BaseEstimator):
7   +           sklearn.utils.estimator_checks.check_
8   +                   estimator(SklearnWrapperClassifier)
```

Fig. 3: Migration for a `sklearn.utils.estimator_checks.check_estimator` deprecated API usage from Scikit-Learn version 0.23.0

- **Change a parameter type:** This operation changes the type of a deprecated API usage parameter. The operation first checks whether the type of the current API argument matches the new type in the updated API. If the type does not match, the deprecated API argument needs to be changed to follow the new type specification, which may require creation of new objects or values. Figure 3 provides an example of this operation for a usage of the `sklearn.utils.estimator_checks.check_estimator` deprecated API. In this example, the allowed parameter type in the updated API is changed to only the `Estimator` type. To ensure that the argument type match the type requirement of the updated API, a type check is added for the function invocation argument (lines 5-8).
- **Add a constraint to a parameter value:** This operation adds a constraint to the value of the API parameter due to a change in the permitted values of the parameter. The API argument's value is checked to ensure that the value fits the newly permitted range of values. If the argument's value does not fit, we need to modify its value accordingly. An example of this update operation is shown in Figure 4 for the `torch.normal` deprecated API. In this example, the value of the `out` argument must have the same size as the `mean` and `std` arguments. To update the usage of this deprecated API, we add an `if` statement that checks the sizes of these objects (line 4-5).
- **Remove API:** This operation removes the deprecated API

```
1       import torch
2       def normal(mean, std, output):
3   -       torch.normal(mean, std, out=output)
4   +       if mean.size()==output.size() and
5   +               std.size()==output.size():
6   +           torch.normal(mean, std, out=output)
```

Fig. 4: Migration for a `torch.normal` deprecated API usage from PyTorch version 1.5.0

```
1       from sklearn.model_selection import KFold
2       def KFold_Selection(seed):
3   +       if seed is not None:
4               KFold(random_state = seed, shuffle=True)
5   +       else:
6   +           KFold(random_state = seed)
```

Fig. 5: Deprecated API migration involving a context-dependent change for `sklearn.model_selection.KFold` from Scikit-Learn version 0.22.0

without replacing its usage with any updated API. This is typically used when the deprecated API is no longer needed or no longer has any effect when invoked.

D2: API Mapping. This dimension represents the ratio between the number of the deprecated and updated APIs involved in the migration. We found three types of mappings:

- **1:1 API mapping:** A deprecated API is modified or replaced by a single updated API.
- **1:N API mapping:** A deprecated API is modified or replaced by at least two updated APIs.
- **1:0 API mapping:** A deprecated API is removed without any suggested replacement.

D3: Context Dependency. This dimension indicates whether the deprecated API migration depends on the context of the deprecated API usage. This context refers to the value and type of the arguments in the deprecated API usage. We found two categories of context dependency.

- **A context-dependent update:** An API migration that depends on the value of one or more arguments of the API invocation. For example, the `sklearn.model_selection.KFold` API is only deprecated if the `random_state` argument is not `None`, in which case the `shuffle` argument must be set to `True`. Otherwise, there is no change to the API. For example, in Figure 5, the update is done by adding an `if` statement that checks the value of the contextual variable `seed` used as the `random_state` argument.
- **A context-independent update:** An API migration that is not affected by the value of the API invocation arguments.

D. RQ2: Distribution of Migration Categories (Types)

In the second research question, we are interested in the distribution of deprecated API migration categories within the dimensions defined in Section II-C: update operation, API mapping, and context dependency. We labeled each of the 112 APIs based on the categories, across the 3 dimensions.

First, we label the APIs based on the update operation. Table II shows the result of this labeling. We found that most deprecated API migrations in Scikit-Learn involve removing a parameter. PyTorch and TensorFlow deprecated API migrations mainly rename methods. TensorFlow API migrations often rename parameters. Among the operations, change parameter type, add constraint to parameter value, and remove API are the least commonly used, amounting to 5 or fewer APIs for each update operation. Note that some API migrations have multiple operations, such as the `tensorflow.compat.v1.to_float` deprecated API which requires both the rename method and add parameter operations.

TABLE II: Distribution of API migrations based on the required update operation to perform the migration

	Scikit-Learn	**PyTorch**	**TensorFlow**
Remove Param	12	1	0
Rename Param	0	1	25
PosToKey Param	0	8	0
Rename Method	4	18	17
Add Parameter	2	1	9
Change Param Type	2	3	0
Add Constraint	1	4	0
Remove API	1	0	3

TABLE III: Distribution of API migrations based on its API mapping

	Scikit-Learn	**PyTorch**	**TensorFlow**
1:1	21	36	50
1:N	0	0	1
1:0	1	0	3

We also label each API based on its API mapping, as shown in Table III. Almost all deprecated API migrations follow the 1:1 transformation, suggesting that API developers prefer a straightforward, less complex API update. Out of the 112 APIs, only 1 API follows the 1:N update category. The 1:0 category is found in 4 APIs.

Then, we label each API based on its update context. The results of this labelling are shown in Table IV. The majority of the API updates are context-independent. Out of the 112 APIs, only 9 API updates are context-dependent.

TABLE IV: Distribution of the API migrations based on the contextual dependency of the API update

	Scikit-Learn	**PyTorch**	**TensorFlow**
Context-independent	19	30	54
Context-dependent	3	6	0

From these distributions within the three dimensions, we found that simpler updates are prevalent among different Python ML libraries. Rename method, rename parameter, add parameter, and remove parameter are the most commonly used, each being observed in the migrations of at least 10 APIs deprecated APIs. 107 out of 112 APIs follow the 1:1 update mapping. Finally, the majority of API updates are context-independent, which accounts for 103 out of 112 APIs. We leverage these findings to guide our priorities in creating an automated update tool for Python ML deprecated API usage.

III. Automated Update for Python Deprecated API Usages

We incorporate the findings of our empirical study in creating an automated update tool for Python deprecated APIs, mainly to direct the types of transformations that the tool should support. We create MLCatchUp, a tool that provides automated updates for Python deprecated API usages. MLCatchUp infers the required transformations for the update by comparing the difference between the deprecated API

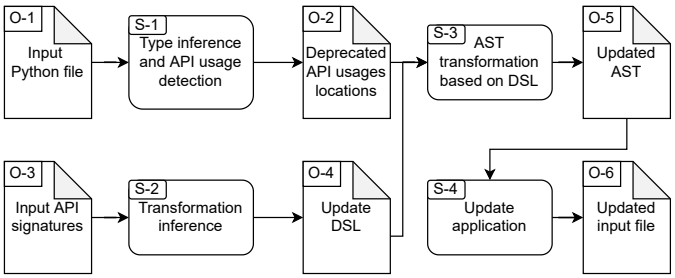

Fig. 6: MLCatchUp Architecture

signature and the updated API signature. Using these API signatures and the code containing the usage of deprecated API as an input, MLCatchUp will automatically provide a new version of the code where all usages of the specified deprecated APIs are replaced with their corresponding updated APIs. During the transformation process, MLCatchUp utilizes a domain specific language to make its transformation more readable for users. MLCatchUp also utilizes type inference to mitigate the problem of dynamic typing that we highlighted in Section I. In the following subsections, we discuss in detail the architecture and features of MLCatchUp.

A. MLCatchUp Architecture

The architecture and pipeline of MLCatchUp are shown in Figure 6. There are two main inputs to MLCatchUp, namely the input Python file (O-1), and the input API signatures (O-3). The input Python file is the file that is going to be updated, while the input API signatures are the deprecated API signature and the updated API signature. The input Python file is transformed into its AST using the abstract syntax tree module provided by Python[6] for easier transformation. From the AST, MLCatchUp uses type inference to determine the types of variables, and the types of method call return values. Then, the deprecated API usages are located by comparing the fully qualified name of the method calls within the code to the deprecated API signature (S-1). This process results in the deprecated API usage locations (O-2).

Input API signatures are used in the transformation inference process (S-2). The transformations are in the form of DSL commands (O-4), containing a series of atomic operations required for the update (e.g. rename method, rename parameter, etc.) which are to be performed sequentially. The DSL is then parsed into a list of operations that can be executed by MLCatchUp. These operations are applied to the deprecated API usage locations (S-3), producing the updated AST (O-5).

The updated AST is then compared with the input file AST to check for code differences between the two ASTs. The code differences are then applied to the input Python file (S-4) by making only the necessary changes to the API usages without any modification to the original code comments and spacing. This results in the updated input file (O-6).

In the following subsections, we present the main components of the MLCatchUp automated update process.

[6]https://docs.python.org/3/library/ast.html

TABLE V: Mapping between each transformation operation and its DSL grammar

Operation	DSL Grammar
Rename Method	$\langle rename\_method \rangle$
Rename Parameter	$\langle rename\_parameter \rangle$
Remove Parameter	$\langle remove\_parameter \rangle$
Pos. to Key. Parameter	$\langle pos\_to\_key\_parameter \rangle$
Add Parameter	$\langle add\_parameter \rangle$
Change Parameter Type	$\langle type\_constraint \rangle$
Add Parameter Constraint	$\langle value\_constraint \rangle$
Remove API	$\langle remove\_api \rangle$

B. Transformation Operations and DSL

MLCatchUp supports the 8 update operations identified in our empirical study. Each operation is paired with its corresponding construct in the DSL to describe the transformation. The required transformation operations for each API migration are inferred automatically, as we will discuss in Section III-C. The inference process produces a DSL script that specifies all the required transformations between a pair of deprecated and updated APIs. The grammar of the DSL can be accessed from our (anonymous) online appendix: https://doi.org/10.6084/m9.figshare.14525136.v3.

Each transformation listed in the DSL grammar corresponds to a basic transformation operation provided by MLCatchUp. The mapping between each basic transformation operation and its DSL grammar is shown in Table V.

C. Transformation Inference

MLCatchUp automatically infers the required change to update the usage of the deprecated API using the deprecated and updated API signatures as its inputs. Based on these signatures, MLCatchUp produces DSL commands representing the steps to update code containing deprecated API usages.

We define an API signature as the fully qualified name of the API and the list of parameters available for the API, where each parameter consists of the parameter name and an optional default value. A fully qualified API name consists of the API module name and the API function name. This information is sufficient as Python does not provide method overloading. The API parameter definition is declared in a comma separated list of positional and keyword parameters.

The difference between the deprecated and the updated API signature is used to automatically infer the required transformation for the API usage update. The pseudocode of the transformation inference is shown in Algorithm 1. Using these deprecated and updated API signatures, several steps are done to produce the DSL commands to update the deprecated API usage. First, if the updated API signature is an empty string, we add a `remove_api` operation (lines 2-3). If the updated API signature is not empty, we compare the fully qualified name (line 5) of the deprecated and updated API signature, then add a `rename_method` operation if the names differ (line 6). Then, we eliminate identical API parameters from both API signatures (line 8), as the same parameters between the two APIs will not affect the API update. Two API parameters are considered identical if they have the same name and position.

Algorithm 1: MLCatchUp automatic inference

Input : $D \in$ deprecated API signature
$U \in$ updated API signature;
Output: Update transformation DSL commands $Cmds$

1 $Cmds \leftarrow null$
2 **if** $U \equiv null$ **then**
3 $\quad Cmds.add(remove\_API(U_{name}))$
4 **else**
5 $\quad$ **if** $Dname \neq U_{name}$ **then**
6 $\quad\quad Cmds.add(rename\_method(D_{name}, U_{name}))$
7 $\quad$ **end**
8 $\quad D, U \leftarrow eliminate\_identical\_param(D, U)$
9 $\quad$ **foreach** *Parameter* $P \in D_{param}$ **do**
10 $\quad\quad$ **foreach** *Parameter* $Q \in U_{param}$ **do**
11 $\quad\quad\quad$ **while** $\exists$ *(*$P_{position} \equiv Q_{position} \wedge P_{name} \neq Q_{name}$*)* **do**
12 $\quad\quad\quad\quad Cmds.add(rename\_parameter(P, Q))$
13 $\quad\quad\quad\quad D, U \leftarrow remove\_used\_param(P, Q)$
14 $\quad\quad\quad$ **end**
15 $\quad\quad\quad$ **while** $\exists$ *(*$P_{name} \equiv Q_{name} \wedge P$ *is a positional param* $\wedge$ Q *is a keyword param)* **do**
16 $\quad\quad\quad\quad Cmds.add(pos\_to\_key\_parameter(P, Q))$
17 $\quad\quad\quad\quad D, U \leftarrow remove\_used\_param(P, Q)$
18 $\quad\quad\quad$ **end**
19 $\quad\quad\quad$ **while** $\exists$ *(*$P \in D_{param} \wedge P \notin U_{param}$*)* **do**
20 $\quad\quad\quad\quad Cmds.add(remove\_parameter(P))$
21 $\quad\quad\quad$ **end**
22 $\quad\quad\quad$ **while** $\exists$ *(*$Q \in U_{param}$*)* **do**
23 $\quad\quad\quad\quad Cmds.add(add\_parameter(Q))$
24 $\quad\quad\quad$ **end**
25 $\quad\quad$ **end**
26 $\quad$ **end**
27 **end**
28 **return** $Cmds$

After removing identical parameters, we compare the remaining API parameters from the two API signatures. Two possible actions are taken during this comparison. First, if two API parameters have the same position but different keyword names, we add a `rename_parameter` operation into the DSL (lines 11-14). Second, if two API parameters have the same name, but one is a positional parameter in the deprecated API signature and the other is a keyword parameter in the updated API signature, we add a `positional_to_keyword_parameter` operation (lines 15-18). Then, we check whether there are any remaining API parameters in the deprecated and the updated API signatures. Each of the remaining API parameters in the deprecated API signature is added as a `remove_parameter` operation into the DSL commands (lines 19-21). Finally, for each remaining API parameter in the updated API signature, we add an `add_parameter` operation (lines 22-24).

To illustrate, consider the `tensorflow.compat.v1.to_float()` deprecated API. This API usage needs to be updated to use the `tensorflow.cast(dtype=tensorflow.float32)` API. MLCatchUp infers the transformation as follows. First, MLCatchUp finds a difference between the fully qualified name of the deprecated API and the fully qualified name of the updated API, and thus a `rename_method` operation is added. Then, MLCatchUp finds that there are no identical parameters between the two API signatures, leaving only a single API parameter in the updated API signature. This parameter is added as

Input:
deprecated: tensorflow.compat.v1.to_float()
updated : tensorflow.cast(dtype:tf.dtypes.DType=tensorflow.float32)

Inferred Transformation:

```
add_parameter dtype WITH_VALUE tensorflow.float32
FOR tensorflow.compat.v1.to_float

rename_method tensorflow.compat.v1.to_float TO
tensorflow.cast
```

Fig. 7: DSL commands produced to convert `tensorflow.compat.v1.to_float()` usages to use `tensorflow.cast(dtype=tensorflow.float32)`

```
1  import tensorflow, tensorflow.compat.v1 as tf
2  + from tensorflow import cast
3  def parse_example(d):
4      img = tf.decode_raw(d['image_raw'], tf.uint8)
5  -   d['image'] = tf.to_float(img)
6  +   d['image'] = cast(img, dtype=tensorflow.float32)
```

Fig. 8: Migration result for `tensorflow.compat.v1.to_float()` deprecated API usage from TensorFlow version 2.0.0

an `add_parameter` operation. The value of the parameter added in the `add_parameter` operation is obtained from the default parameter value listed in the API signature. The produced DSL commands are shown in Figure 7.

These DSL commands can be applied to files using the deprecated `tensorflow.compat.v1.to_float()` API. First, the file is parsed into its AST. Then, each DSL command is parsed and translated into the corresponding transformation that is applied to the AST. After the transformations are applied, the AST is changed back into code, which is then compared with the deprecated code. Based on the code differences, MLCatchUp applies the transformation to the original code, making only the necessary changes towards the deprecated API usages, retaining the comments and spacing of the original code. An example result of the update is shown in the code *diff* in Figure 8.

While MLCatchUp can infer the required transformation operations, MLCatchUp is unable to infer the required information regarding any added constraints for the transformation itself. Indeed the API signatures do not provide such information. To mitigate this problem, we allow MLCatchUp to be run with an optional argument in which the user can specify the transformation constraint as an input. This input constraint must be written using the MLCatchUp DSL. An example of the inferred DSL script and the provided input constraint for the `sklearn.model_selection.KFold` API update is shown in Figure 9.

In Figure 9, we see that from the API signatures, MLCatchUp is able to infer that the uses of the `sklearn.model_selection.KFold` deprecated API need to be updated with the addition of the `shuffle=True` parameter. However, MLCatchUp is unable to obtain the transformation constraint from the API signatures, providing an incomplete

Input:

deprecated: sklearn.model_selection.KFold(random_state:int=0)
updated : sklearn.model_selection.KFold(random_state:int=0, shuffle:bool=True)

Inferred Transformation:

```
add_parameter shuffle WITH_VALUE True FOR
sklearn.model_selection.KFold
```

Input Constraint

```
IF random_state HAS VALUE != None
```

Fig. 9: DSL commands with input constraint for migrating `sklearn.model_selection.KFold` deprecated API usage from Scikit-Learn version 0.23.0 to add `shuffle=True` API parameter

```
1  import torch
2  x1=torch.empty(5,3)
3  x2=torch.rand(5,3)
4  print(x1.add(x2))
```

Fig. 10: Example of `torch.Tensor.add` API usage

inferred series of transformation commands. In reality, the add parameter update operation should only be done if the value of `random_state` parameter is not None. This constraint can be expressed through a user input constraint.

D. Type Inference

In MLCatchUp, we incorporate type inference to determine the types of variables and the return types of method calls within the code. This type information can then be used to substitute the variable or the method invocation used in an API invocation to determine the API's fully qualified name. As an example, given the code shown in Figure 10, MLCatchUp will produce the variable and method call type information shown in Table VI. By using the inferred type information in the `x1.add(x2)` API invocation in line 4, we replace the variable `x1` by the inferred type `torch.Tensor`, which results in the API invocation of `torch.Tensor.add(x2)` with the fully qualified API name `torch.Tensor.add`. This replacement is only done as an internal process of MLCatchUp to determine the fully qualified name of API invocations in the processed code.

We implement this type inference by utilizing the PEP (Python Enhancement Proposal) 484,[7] which focuses on the usage of type hints and type annotations for Python code. Following this PEP, third-party libraries have started to provide type annotations in their library code, providing hints on the return type of the library's API methods. In MLCatchUp, we make use of Mypy [11], a static type checker for Python that uses the PEP 484 type hints. Given a Python code file as an input, we use Mypy's API to analyze the code, collecting the types of variables and method calls by searching through the imported library for type annotations. We then use these collected types to replace the variables and method calls during the deprecated API usage detection.

[7]https://www.python.org/dev/peps/pep-0484/

TABLE VI: MLCatchUp's inferred type from the code example in Figure 10

Expression	Variable/Method Call	Inferred Type
x1	Variable (line 2)	torch.Tensor
x2	Variable (line 3)	torch.Tensor
torch.empty	Method call	torch.Tensor
torch.rand	Method call	torch.Tensor
x1.add	Method call	torch.Tensor

One caveat of this approach is that it only works for a library that provides type annotations. To mitigate this problem, we further enhance our type inference functionality by utilizing deep learning-based type inference. Specifically, we use Type4Py [10] that implements a deep similarity learning approach to infer the return type of a method invocation. Type4Py is trained on a dataset containing 4,910 Python projects with more than 1.1 million type annotations. From the source code of these projects, Type4Py extracts features from method invocations of known types. These features are extracted from the method names and the arguments appearing in the invocations, the body of the corresponding method definitions, as well as relevant import statements. These features are then used to train a deep learning model that can map a method invocation to its semantic vector representation. Next, given a method invocation with an unknown type, Type4Py infers its type by: (1) converting the method invocation to its semantic vector, (2) identifying k type-annotated data instances in the training set with similar semantic vectors (by default, k is set to 10), and (3) identifying the most common types of the k data instances. MLCatchUp uses Type4Py's top-most predicted type in the same way as the type inferred using Mypy.

IV. MLCatchUp Evaluation

A. Dataset Collection

Based on our empirical study, we use deprecated APIs from the Scikit-Learn, TensorFlow, and PyTorch libraries for our evaluation. We consider the 112 identified deprecated APIs from the three libraries. To further evaluate the performance and generalizability of MLCatchUp on additional ML libraries, we also investigate 8 deprecated APIs from Spacy and Numpy, which are Python libraries that are typically used for natural language processing and array processing. In order to provide a realistic evaluation, we use real API usages. With this in mind, we directed our search towards public GitHub repositories.

For the evaluation data collection, we created a search tool adapted from the work of Wang et al. [2]. Our search tool makes use of the GitHub search API, allowing up to 1,000 repositories to be retrieved for each search query. Each fully qualified name of API invocation is matched with the deprecated API signature to detect usages of deprecated APIs in the code. Using the deprecated API signatures as the input, we collected our evaluation data.

B. Evaluation Dataset

Using our search tool, we collected code containing deprecated API usages from GitHub public repositories for the 120

APIs (112 APIs from Scikit-Learn, PyTorch, and Tensorflow + 8 APIs from Numpy and Spacy). Out of the 120 APIs, we found public code usages for only 68 APIs. For each of these 68 APIs, we collected at most five files containing deprecated API usage for our evaluation dataset. For deprecated APIs used by more than five files, we randomly selected five files. Meanwhile, for deprecated APIs used by only five or fewer files, we used all of the files for our evaluation. In total, we collected 267 files containing 551 API usages from 68 different APIs. 44 files contain Scikit-Learn API usages, 82 files contain TensorFlow API usages, 110 files contain PyTorch API usages, 22 files contain Spacy API usages, and 9 files contain Numpy API usages.

C. Experiment Setting

We conducted an experiment to evaluate the ability of MLCatchUp to update the usages of deprecated APIs. We aim to measure whether MLCatchUp can detect deprecated API usages, and to check whether MLCatchUp is able to correctly update their usages. With this goal in mind, we calculated the *detection rate* of the deprecated API usages. We define the detection rate as the amount of correctly detected deprecated API usages divided by the number of deprecated API usages in the ground truth. We also calculated the *accuracy* of the update by measuring the number of correct updates produced compared to the number of deprecated API usages.

To check the update results, we make use of automated and manual analysis of the updated code. In the automated analysis, we automatically check whether the updated code produces any syntax error by compiling the updated code using the Python built-in py_compile module.[8] In the manual analysis, we employed the help of three Python programmers with at least four years of experience to label whether the updated code is correct. These programmers are non-authors, and were given instruction and training before the labelling process. Each instance of the deprecated API usages is labelled by two different programmers. If any labelling disagreement arises between the two programmers, we conducted a discussion until a consensus is achieved. Throughout the labelling process, only two disagreements occured, which resulted in 99.6% inter-rater reliability. We also measure the mean pairwise Cohen's kappa [14] of the labelling result from the three programmers, and achieve a mean pairwise Cohen's kappa of 0.986, indicating a near perfect agreement.

D. Deprecated API Usage Detection Evaluation

For the deprecated API usage detection evaluation, we performed two different measurements: (1) detection rate without type inference, and (2) detection rate with type inference. The result of these measurements are provided in Table VII. From the result, we observe that without the type inference, MLCatchUp is unable to detect all of the deprecated API usages, achieving a detection rate of 93.7%. The integration of type inference in MLCatchUp, improves its detection rate, as shown by MLCatchUp achieving a 100.0% detection rate.

[8]https://docs.python.org/3/library/py_compile.html

TABLE VII: MLCatchUp detection rate evaluation results on all data

Evaluation	Detected	Ground Truth	Detection Rate
No Type Inference	516	551	93.7%
Type Inference	551	551	100.0%

TABLE VIII: MLCatchUp detection rate evaluation results on Numpy and Spacy deprecated API usages

Evaluation	Detected	Ground Truth	Detection Rate
No Type Inference	25	57	43.9%
Type Inference	57	57	100.0%

We further check whether MLCatchUp detection rate holds for the Spacy and Numpy ML libraries. For this purpose, we conduct the same deprecated API usage evaluation only on these two additional libraries. The result of this evaluation is shown in Table VIII. From the result, we observe that without type inference, MLCatchUp has a low detection rate of 43.9%. With the addition of the type inference, it also manages to achieve 100.0% detection rate of the deprecated API usages.

E. Update Accuracy Evaluation

In the update accuracy measurement, we conducted two experiments, (1) experiment without user-added constraints, and (2) experiment with user-added constraints. The results of both experiments are shown in Table IX.

From the experiment without added constraints, MLCatchUp achieved an accuracy of 80.6%. There are 19.4% instances of updated code that are labelled as false positives, indicating deprecated API usages that were updated incorrectly by MLCatchUp. One of the main reasons for the presence of false positives is due to the transformation constraint not being present for context-dependent updates because such a constraint cannot be automatically inferred by MLCatchUp.

To address the problem of context-dependent migration, we add user-added constraints for context-dependent deprecated API usage updates. Using this additional input, we conducted the follow-up experiment on MLCatchUp using the same test dataset. We manually define the user-added constraints for each context-dependent deprecated API usage migration and add them as an input to MLCatchUp. In total, we manually write 7 constraints. The result of this follow-up experiment is shown in the third row of Table IX.

Using the user-added constraints, MLCatchUp achieves 90.7% accuracy, which is an improvement of more than 10 percentage points compared to the previous experiment. While the addition of the user-added constraints improves the overall accuracy of MLCatchUp update result, there remain some incorrect updates. MLCatchUp failed to correctly update 9.3% of the deprecated API usages. To understand the reasons for

TABLE IX: MLCatchUp update accuracy evaluation results on all data

Experiment	Correct	Incorrect	Accuracy
Without added constraints	444	107	80.6%
With added constraints	500	51	90.7%

TABLE X: MLCatchUp update accuracy evaluation results on Numpy and Spacy deprecated API usages

Experiment	Correct	Incorrect	Accuracy
Without added constraints	53	4	93%

```
1  - torch.addcdiv(input, tensor, other)
2  + input + torch.floor_divide(tensor, other)
```

Fig. 11: Migration for `torch.addcdiv` deprecated API usage from PyTorch version 1.6.0

these failures, we conducted a qualitative study, which we will discuss in the next subsection.

We also analyze whether MLCatchUp's update accuracy also applies to the additional ML libraries. We found that APIs from these libraries are context-independent, which mean that they do not require user-added constraints for the update. We conduct the update accuracy evaluation on the Numpy and Spacy deprecated API usages. The result of this evaluation is shown in Table X. We observe that MLCatchUp achieves 93.0% update accuracy on the additional ML libraries deprecated API usages. This indicates that MLCatchUp automatic update generalize well to other ML libraries.

F. MLCatchUp's Failed Migrations

Based on our evaluation of MLCatchUp, we see that 51 instances of deprecated API usage are updated incorrectly. We analyze these cases to find the reasons for these failed migrations. The reasons are:

1) *API usage migration involving arithmetic operations*. Deprecated API usage migration may involve an addition of arithmetic operations between the updated API and other values, such as the deprecated API parameter. This is currently not handled by MLCatchUp. This case is found in the `torch.addcdiv` deprecated API migration, shown in Figure 11. In this example, the deprecated API usage (line 1) needs to be replaced by an addition arithmetic operation between the `input` parameter of the deprecated API usage and a `torch.floor_divide` API invocation (line 2). In the future, we can add support for arithmetic operation to handle this type of API migration.
2) *API usage migration involving a 1:N API mapping*. Currently, MLCatchUp only handles API usage migrations with 1:1 and 1:0 API mappings, which are the most common API mapping migration categories identified in our empirical study (see Section II). An example of the 1:N case is found in the `tensorflow.compat.v1.sparse_to_dense` deprecated API migration, where the update requires adding an invocation of `tensorflow.sparse.SparseTensor` and an API name change from `tensorflow.compat.v1.sparse_to_dense` to `tensorflow.sparse_to_dense`. This API migration is illustrated in Figure 12. In the future, it is possible to extend the operations provided by MLCatchUp to include operations adding new API invocations.

```
1  - indices_scattered = tf.compat.v1.sparse_to_dense(
2  -       sparse_indices, tf.cast(tf.shape(shape),
3  -       tf.int64), indices_gathered)
4  + sparse_tensor = tf.sparse.SparseTensor(sparse_indices,
5  +       indices_gathered, tf.cast(tf.shape(shape),
6  +       tf.int64))
7  + indices_scattered = tf.sparse.to_dense(sparse_tensor)
```

Fig. 12: Migration for `tensorflow.compat.v1.sparse_to_dense` deprecated API usage from TensorFlow version 2.0.0

```
1      import torch
2      def MAPE_torch(pred, mask):
3  +       mask = torch.gt(true, mask - 0.01)
4          pred = torch.masked_select(pred, mask)
```

Fig. 13: Migration for `torch.masked_select` deprecated API usage from PyTorch version 1.2.0

3) *API usage migration involving value and/or type modifications of the parameter*. MLCatchUp addresses the problem of parameter value or type change through the usage of transformation constraints. However, this approach can be insufficient as some deprecated APIs require the parameter values to be updated, which typically include constructing a new value through mathematical operations or other API invocations. An example of this case is found in the `torch.masked_select` deprecated API usage, which is shown in Figure 13. In the deprecated version, `torch.masked_select` allows for `IntTensor` as its second positional parameter. In the updated version, the second positional parameter must be of type `BoolTensor`, hence the conversion is done using the `torch.gt` API in line 3. In the future, it is possible to add more operations into MLCatchUp that would enable modifications of the values of API parameters.

V. Threats to Validity

Threats to *external validity* relate to the generalizability of our findings. Our research investigated 112 APIs from three popular Python machine learning libraries, namely Scikit-Learn, TensorFlow, and PyTorch. We made this pragmatic choice to address the recent pain points that many developers face. We designed MLCatchUp based on the results of our empirical study, which may not hold true for other libraries. To mitigate this threat, we extend our evaluation dataset by including 8 additional deprecated APIs from Spacy and Numpy. In our evaluation, we find that MLCatchUp is capable of creating correct updates for Spacy's and Numpy's API migrations. Considering this result, we believe that there is minimal threat for this potential issue.

Threats to *external validity* also relate to our approach in collecting API deprecation data. We conduct our deprecated API collection by focusing on the change logs of the three libraries appearing in their official documentations and/or release notes. We understand that using this approach, we may miss some deprecated APIs that are not documented, as has

been highlighted by Wang et al. [2]. Nevertheless, currently there is no standard in the documentation of Python library deprecated APIs and developers tend to use ad-hoc strategies to deprecate APIs. As such, we believe that the libraries' official documentations and/or release notes are still the most complete source of the API deprecation data.

Threats to *internal validity* relate to MLCatchUp's evaluation. We measure the update accuracy of MLCatchUp by checking the syntax of the produced code and through manual analysis by asking the help of three experienced Python programmers to determine the correctness of the updated code. It is possible that false positives (i.e., incorrectly updated code labelled as correct) or false negatives (i.e., correctly updated code labelled as incorrect) may occur due to human error. To mitigate this risk, we have shared a replication package of our evaluation. Using this replication package, others can check and improve the labels used in our accuracy evaluation.

VI. Related Work

API deprecation: Zhou and Walker [15] created a framework to detect deprecated API usages in source code examples for Android. Brito et al. [16] proposed a recommendation tool that helps to find deprecated API alternatives by mining solutions adopted by developers. Sawant et al. [17], [18] found that most developers do not update their API usages due to the cost. Xavier et al. [19] found that the frequency of API deprecations increases over time and more popular systems have a higher frequency of deprecations. Li et al. [20] characterized deprecated Android APIs, finding that usage of deprecated APIs is mostly found in popular libraries. Our study is focused on the API deprecation of Python ML libraries, which has not been investigated before.

API migration: apiwave [21] keeps track of API popularity and migration of major frameworks and libraries from top 650 GitHub Java projects. Diff-CatchUp [22] is a classic work that automatically detects API changes in a Java library, infers possible replacements of the changed APIs, and identifies possible examples of how to migrate the changed APIs. Different from our approach, Diff-CatchUp does not produce a transformation script or comes with a transformation engine, which provides automatic update for the broken API usages in the client code base. Thus, developers utilizing Diff-CatchUp need to manually perform the transformation based on the change examples.

Program transformation: Sydit [23] generates a context-aware, abstract edit script using only one example. LASE [24] automatically transforms code by creating context-aware edit scripts from two or more examples and use the scripts to automatically identify edit locations to be transformed. REFAZER [25] automatically learns program transformations from observation of code edits performed by developers. Using input-output examples, REFAZER synthesizes program transformations for Python code. SPINFER [26] is capable of automatically inferring Coccinelle semantic patches from existing code change examples for the Linux kernel by leveraging similar code fragments and control flows in the code changes to identify the patterns.

A4 [6] assists Android API migration by learning API migration patterns from source code examples. AppEvolve [7] transforms the usage of deprecated Android APIs into backward-compatible updated code by learning from code examples. CocciEvolve [8] provides improvement from AppEvolve by creating readable transformations using only a single update example. Thung et al. [9] proposed NEAT, a tool to generate transformation rules that can assist developers in deprecated API replacement for Android APIs.

There are a number of differences between MLCatchUp and the aforementioned program transformation works. First, none of the previous works supports Python, with the majority supporting Java and C. Also, different from most of the prior works, MLCatchUp does not require any code example to infer transformations to migrate deprecated API usages. Moreover, MLCatchUp presents its transformation in the form of a DSL that is created based on the result of our empirical study of Python ML libraries. Furthermore, MLCatchUp performs type inference to address the problem associated with Python being a dynamically-typed language.

VII. Conclusion and Future Work

We have conducted an empirical study using a collected list of 112 deprecated APIs to learn the characteristics of the API migrations required by Python deprecated APIs. Using manual thematic analysis, we found a number of categories of API considering three dimensions: update operation, API mapping, and context dependency. Based on findings in our empirical study, we created MLCatchUp, an automated tool to update the usage of deprecated ML APIs in Python. MLCatchUp automatically infers the transformation operations required to update the usage of a deprecated API. MLCatchUp utilizes type inference to deal with the fact that Python is a dynamically-typed language. We have evaluated MLCatchUp using a dataset of 267 files containing 551 API usages from 68 different APIs, where it manages to detect 93.7% of the deprecated API usages without type inference. Utilization of type inference improves this detection rate to 100.00%. For the update accuracy measurement MLCatchUp achieves 80.58% accuracy. We have further improved MLCatchUp's accuracy by adding transformation constraint inputs, allowing it to handle context-dependent deprecated API usage updates. With this improvement, MLCatchUp achieves 90.74% accuracy.

For future work, we plan to extend our study to other popular Python libraries. By increasing the scope of the case study, we may encounter more variety of deprecated APIs and their migrations. We also plan to address the current shortcomings of MLCatchUp, which have been described in Section IV-F. A replication package of MLCatchUp can be accessed in the following link: https://zenodo.org/record/4736246

Acknowledgement. This research is supported by the Singapore NRF (award number: NRF2016-NRF-ANR003) and the ANR ITrans project.

REFERENCES

[1] J. Han, S. Deng, D. Lo, C. Zhi, J. Yin, and X. Xia, "An empirical study of the dependency networks of deep learning libraries," in *ICSME*, 2020.

[2] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *FSE*, 2020.

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

[4] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, p. 291–300. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00042

[5] F. Khomh, B. Adams, J. Cheng, M. Fokaefs, and G. Antoniol, "Software engineering for machine-learning applications: The road ahead," vol. 35, no. 5, 2018, pp. 81–84.

[6] M. Lamothe, W. Shang, and T.-H. Chen, "A4: Automatically assisting Android API migrations using code examples," *ArXiv*, vol. abs/1812.04894, 2018.

[7] M. Fazzini, Q. Xin, and A. Orso, "Automated API-usage update for Android apps," in *ISSTA*. ACM, 2019, pp. 204–215.

[8] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic Android deprecated-API usage update by learning from single updated example," in *IEEE International Conference on Program Comprehension*, 2020.

[9] F. Thung, H. Kang, L. Jiang, and D. Lo, "Towards generating transformation rules without examples for Android API replacement," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 09 2019, pp. 213–217.

[10] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4py: Deep similarity learning-based type inference for python," *arXiv preprint arXiv:2101.04470*, 2021.

[11] J. Lehtosalo, G. v. Rossum, I. Levkivskyi, M. J. Sullivan, D. Fisher, G. Price, M. Lee, N. Seyfer, R. Barton, S. Ilinskiy, E. Smith, and J. Zijlstra, "Mypy - optional static typing for python," https://github.com/charlespwd/project-title, 2017.

[12] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, pp. 77–101, 01 2006.

[13] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 275–284.

[14] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[15] J. Zhou and R. Walker, "API deprecation: a retrospective analysis and detection method for code examples on the web," 11 2016, pp. 266–277.

[16] G. Brito, A. Hora, M. Valente, and R. Robbes, "On the use of replacement messages in API deprecation: An empirical study," *Journal of Systems and Software*, vol. 137, 12 2017.

[17] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli, "Why are features deprecated? an investigation into the motivation behind deprecation," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 13–24.

[18] A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs," 10 2016, pp. 400–410.

[19] L. Xavier, A. Brito, A. Hora, and M. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 02 2017, pp. 138–147.

[20] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated Android APIs," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. ACM, 2018, pp. 254–264.

[21] A. Hora and M. T. Valente, "Apiwave: Keeping track of API popularity and migration," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 321–323.

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

[23] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 329–342.

[24] J. Jacobellis, N. Meng, and M. Kim, "LASE: An example-based program transformation tool for locating and applying systematic edits," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1319–1322.

[25] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *ICSE*. IEEE Press, 2017, pp. 404–415.

[26] L. Serrano, V.-A. Nguyen, F. Thung, L. Jiang, D. Lo, J. Lawall, and G. Muller, "SPINFER: Inferring semantic patches for the Linux kernel," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 235–248.