How Do Python Framework APIs Evolve?
An Exploratory Study

Zhaoxu Zhang*, Hengcheng Zhu*, Ming Wen†, Yida Tao‡, Yepang Liu*, and Yingfei Xiong§

* Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China.
{11611308, zhuhc2016}@mail.sustech.edu.cn, liuyp1@sustech.edu.cn
† School of Cyber Sci. and Engr., Huazhong Univ. of Sci. and Tech., Wuhan, China. mwenaa@hust.edu.cn
‡ College of Comp. Sci. and Soft. Engr., Shenzhen University, Shenzhen, China. yidatao@szu.edu.cn
§ Department of Computer Science and Technology, EECS, Peking University, Beijing, China. xiongyf@pku.edu.cn

Abstract—**Python is a popular dynamic programming language. In recent years, many frameworks implemented in Python have been widely used for data science and web development. Similar to frameworks in other languages, the APIs provided by Python frameworks often evolve, which would inevitably induce compatibility issues in client applications. While existing work has studied the evolution of frameworks in static programming languages such as Java, little is known on how Python framework APIs evolve and the characteristics of the compatibility issues induced by such evolution. To bridge this gap, we take a first look at the evolution of Python framework APIs and the resulting compatibility issues in client applications. We analyzed 288 releases of six popular Python frameworks from three different domains and 5,538 open-source projects built on these frameworks. We investigated the evolution patterns of Python framework APIs and found that they largely differ from those of Java framework APIs. We also investigated the compatibility issues in client applications and identified common strategies that developers adopt to fix these issues. Based on the empirical findings, we designed and implemented a tool, PYCOMPAT, to automatically detect compatibility issues caused by misusing evolved framework APIs in Python applications. Experiments on 10 real-world projects show that our tool can effectively detect compatibility issues of developers' concern.**

Index Terms—**Python, API evolution, compatibility, dynamic programming language**

I. INTRODUCTION

As the fastest-growing programming language in recent years, Python has reached a new peak in 2019's TIOBE [1] and PYPL [2] index of the popularity of programming languages, ranking the third and first place, respectively. The astonishing growth of Python is largely attributed to the rising interests and needs in data analytics, deep learning, and web services [3]. Respective Python libraries such as *Pandas*, *TensorFlow* and *Django* are all extremely popular and used by hundreds of thousands of client applications.

Similar to frameworks implemented in other languages, the frameworks implemented in Python (Python frameworks for short) also evolve due to reasons such as feature additions, bug fixes, and performance enhancement. Existing work has pointed out that framework evolution may induce compatibility issues in client applications. For example, Dietrich

et al. [4] studied the evolution problems in Java libraries.[1] Wei et al. [5] studied the compatibility issues induced by Android API evolution. However, existing work focused on the evolution of frameworks implemented in *static programming languages* (SPLs) such as Java. Python, as a typical *dynamic programming language* (DPL), differs largely from SPLs. It is unclear whether existing findings based on SPL frameworks can be generalized to Python as well. For example, in the post 38739422 of Stack Overflow [6], users discussed how the removal of an optional parameter `context_instance` in the API `render_to_response` of Django leads to a crash when keyword argument is used to invoke the API. This Stack Overflow post has 62 upvotes and 28,393 views, indicating the profound impact of such issues among Python developers. Unfortunately, such important issues caused by API evolution of Python frameworks cannot be covered by existing studies on SPL frameworks. This motivates us to take a first look at the evolution of Python frameworks and the resulting compatibility issues. Specifically, we aim to answer the following research questions in this work:

- **RQ1 (Framework API Evolution Patterns):** *How do Python framework APIs evolve? What are the common patterns? How do such patterns differ from the API evolution patterns of SPL frameworks?*
- **RQ2 (Compatibility Issues & Fixing Strategies):** *What types of compatibility issues could Python framework API evolution incur in client applications? How do application developers fix such compatibility issues?*

To answer these questions, we analyzed 288 releases of six popular Python frameworks from three different domains and 5,538 GitHub [7] projects that are built on these frameworks. By exploring this dataset, we have made several interesting findings. First, we observed 14 API evolution patterns for Python frameworks, 5 of which were not observed in the evolution of Java frameworks [8]. These five distinct patterns are associated with changes to optional parameters and default parameter values, which are not supported by Java. Second, we found that the percentage of breaking changes during the evolution of Python frameworks is significantly higher than that of Java frameworks. Third, by studying 409 compatibility

---

* The first two authors contributed equally to this work. Yepang Liu is the corresponding author.

[1]We use "framework" and "library" interchangeably in this paper.

SANER 2020, London, ON, Canada
Research Papers

issues, we found that Python applications using evolved APIs often suffer from crashes caused by over 10 types of runtime exceptions, many of which are specific to Python. We further studied the patches of these compatibility issues, and observed that Python developers mainly adopt four kinds of strategies to fix such issues caused by the evolution of framework APIs.

Based on our empirical findings, we designed and implemented a tool, PYCOMPAT, to automatically detect potential compatibility issues caused by misusing evolved framework APIs in Python applications. The tool is powered by a knowledge base that encodes the evolution history of Python framework APIs. It performs light-weight static analyses at the call sites of evolved APIs and applies backward slicing to determine if compatibility issues could arise due to potential API misuses. To evaluate the tool, we applied it on 10 Python applications that use TensorFlow. PYCOMPAT detected compatibility issues at 261 API call sites in these projects, 231 (88.5%) of which are true positives. We further reported these issues to the projects' developers. So far, the developers of four projects have confirmed our reported 74 issues and fixed them quickly.

To summarize, this paper makes four major contributions:

- To our best knowledge, we performed the first systematic study to characterize the evolution of Python framework APIs. We also discussed the similarities and differences between the API evolution in Python and Java frameworks.
- We quantitatively and qualitatively analyzed the types of compatibility issues caused by misusing evolved APIs in Python applications and the commonly adopted fixing strategies. Our findings may help application developers to cope with framework evolution issues more effectively.
- We built a dataset, which contains the API evolution histories of six popular Python frameworks, the compatibility issues, and the corresponding fixes in Python applications. We released the data to facilitate future research (https://doi.org/10.5281/zenodo.2756358).
- We designed and implemented a tool, PYCOMPAT, to detect compatibility issues in Python applications. Experiments on 10 real-world projects show that our tool can effectively detect issues of developers' concern.

II. BACKGROUND

A. The Python Programming Language

Dynamic programming languages (DPLs) are a class of high-level programming languages which, at runtime, execute many common program behaviors (e.g., extending objects and definitions) that static programming languages (SPLs) perform during compile time [9]. Python is a popular DPL. Python programs are executed by an interpreter. When the interpreter translates a .py source file to a .pyc file, which contains byte code, it performs little checking. Due to this reason, problems caused by violating type compatibility rules, calling missing APIs, and even syntax errors will not manifest until runtime.

Argument passing in Python is flexible. When calling a Python function, arguments can be passed as either positional

TABLE I
SELECTED FRAMEWORKS

Framework	Category	# Releases	Versions
TensorFlow [16]	Deep learning	67	0.12.0rc0 – 2.0.0a0
Keras [17]		35	1.0.2 – 2.2.4
scikit-learn [18]	Data analytics	24	0.15.0b1 – 0.20.3
Pandas [19]		42	0.9.1 – 0.24.2
Flask [20]	Web development	12	0.10 – 1.0.2
Django [21]		108	1.8 – 2.2rc1

or keyword arguments [10]. A default value can also be specified for a parameter. It is not compulsory to pass arguments for such parameters at call sites.

Python has no access modifiers. Python developers often follow a widely-adopted convention to make the name of class members that should be treated as "private" start with underscores. However, there is no language-level mechanism to prevent accessing such "private" members of library modules.

B. Library Referencing and Evolution

Libraries ease application development [11]. Many popular libraries frequently evolve. Due to such evolution, an application may encounter different versions of its referenced libraries when it is launched in different environments [11].

Despite the existence of tools for creating isolated environments (e.g., virtualenv [12]) for Python applications, many Python libraries are installed with pip [13], which downloads packages from an online software repository PyPI [14] and installs them for system-wide access. In such cases, libraries might not be updated simultaneously with applications. It is common for a Python program to rely on a certain version of a library but to run in an environment with an older version of that library. This could result in various compatibility issues.

C. Breaking Changes and Non-Breaking Changes

API changes can be *breaking* or *non-breaking* according to their consequences [15]. Breaking changes are backward-incompatible and would cause compilation or runtime problems in client code. For example, signature changes in Java APIs would cause compilation errors. Non-breaking changes (e.g., performance enhancement) are backward compatible and typically would not cause perceivable issues in client code.

III. SUBJECT SELECTION

A. Framework Selection

Answering RQ1 requires us to study real-world Python frameworks. To select such frameworks, we searched on GitHub, a popular open-source project hosting site, with the keyword `topic:python`. We sorted the returned results by the number of stars, which is an indicator of popularity. We checked the top projects and found that they mainly fall into three categories: (1) deep learning, (2) data analytics, and (3) web development. We then selected the top two projects in each of these categories for analysis. Table I lists the projects.

B. Client Applications Selection

Answering RQ2 requires us to study the change histories and bugs of the projects built on our selected Python frameworks. To collect such projects, we searched on GitHub using the keywords `topic:<framework>` and `language:python` to retrieve repositories that reference each of the six frameworks as listed in Table I. For each framework, we selected the top 1,000 repositories with the most stars. By this process, we obtained a set of 5,538 unique repositories. The number is not equal to 6,000 ($1,000 \times 6$) mainly because of two reasons. First, some repositories use more than one of our selected frameworks. Second, there are less than 1,000 projects using *scikit-learn* on GitHub by our searching criteria.

IV. FRAMEWORK API EVOLUTION PATTERNS

A. API Extraction Approach

To answer RQ1, we need to analyze the evolution of the APIs provided by our selected frameworks. As shown in Table I, we collected 288 releases for subsequent analysis.

Extracting Python library APIs is non-trivial.[2] Different from many SPLs, Python objects such as functions can be dynamically created and altered at runtime. In practice, Python developers often leverage this feature to map long fully qualified names of certain APIs to shorter ones. For example, in TensorFlow, `tf.python.ops.array_ops.concat` is mapped to `tf.concat`. Such mapped APIs with shorter names (usually recommended by library developers[3]) cannot be extracted via static analysis since they are not available until runtime. To address this issue, we extracted APIs including fields, methods, and classes dynamically using reflection.[4]

Specifically, we downloaded our selected framework releases from PyPI [14], a repository that indexes millions of Python frameworks. For each release, we imported the library, and then performed a breadth-first search starting from the root module, aiming to extract all the public APIs mentioned above. This process was run in containers using Docker [22], which isolates the API extraction from the configuration of a specific physical machine. Besides, containers enable us to speed up the extraction by running it in parallel on a multi-processor server. Note that we need to analyze hundreds of library releases, each of which may expose thousands of APIs. After this, we obtained a dataset containing the public APIs in each release of our selected frameworks.

B. Evolution Patterns

To understand API evolution patterns, for each of the six frameworks, we first compared the extracted APIs of every two consecutive releases based on information like name and signature. By such comparisons, we found 1,094,359 API changes. In this study, we mainly focus on syntactic changes instead of semantic changes (e.g., the behavior of a method

---

[2]We did not extract APIs from API documentations because we found that they are often unstructured, incomplete, and outdated.

[3]`tf.concat` is in TensorFlow documentation while the longer one is not.

[4]We use "methods" in this paper to refer to functions in a class as well as those out of class. The same applies to "fields".

---

changes but the method signature remains unchanged), since identifying the latter requires program comprehension, which is hard to automate. Without an automated process, it is impractical to analyze over one million changes. Next, we developed a set of matching rules based on nine known API evolution patterns [8], which are listed in Table II (patterns 1–7 and 13–14). These rules helped classify 1,064,081 of the 1,094,359 changes into the nine patterns. For the remaining 30,278 unclassified API changes, we performed random sampling and found that they are related to optional parameters. Accordingly, we composed two matching rules for the addition/removal of optional parameters (patterns 8–9 in Table II) and another three rules for the changes of parameter default values, which may turn a required parameter into an optional one or the other way around (patterns 10–12 in Table II). Using these five rules, we were able to classify all of the remaining 30,278 API changes.

In total, we identified 14 API evolution patterns in our studied Python frameworks, five of which have not been observed in Java frameworks according to a recent work [8]. Table II presents the results, including the frequency of each pattern. In the following, we discuss these patterns in detail.

API addition and removal (96.3%). Most API changes are additions (49.0%) and removals (47.3%) of classes, methods, and fields. As described in Section II-C, API additions are non-breaking changes while API removals are breaking changes. These API changes happened mainly because framework developers implemented new features or cleaned up deprecated features. We also observed that some API removals and additions happened because developers relocated or renamed certain API elements. For example, in Pandas 0.18.0, `pd.rolling_mean` was relocated and then renamed to `pd.DataFrame.rolling`. However, without project-specific domain knowledge or high-quality changelogs, it is hard to dig out such reasons behind API changes.

Parameters changes (3.7%). The remaining API changes are related to method parameters, which are considered as breaking changes [8]. As listed in Table II, we observed eight change patterns in this category, three of which also apply to Java (patterns 5–7) while the other five are specific to Python (patterns 8–12). We discuss the patterns below. Note that the percentages presented below are with respect to parameter changes instead of all API changes.

(1) **Addition and removal of required parameters.** Similar to Java, adding (12.8%) and removing (8.2%) required parameters are also observed in our six frameworks.

(2) **Reordering parameters.** We observed 840 (2.1%) cases of parameter reordering. For instance, in Keras 2.0.6, the order of the parameters `output` and `target` in `keras.backend.binary_crossentropy` was inverted to be consistent with other APIs. Such changes could crash client code if positional arguments are used in API calls (e.g., `foo(1, 2)`).

While the above parameter changes can also happen during the evolution of Java frameworks, the following patterns cannot. Such changes occurred because Python supports default parameter values in method declarations.

TABLE II
API EVOLUTION PATTERNS AND MATCHING RULES

Index	Element	Pattern	Frequency	Matching Rule	Change Type
1	Class (c)	Class Removal*	25,481 (2.3%)	$c \in C \land c \notin C'$	Breaking
2		Class Addition*	26,694 (2.4%)	$c \notin C \land c \in C'$	Non-Breaking
3	Method (m)	Method Removal*	309,759 (28.3%)	$m \in M \land m \notin M'$	Breaking
4		Method Addition*	319,618 (29.2%)	$m \notin M \land m \in M'$	Non-Breaking
5		Required Parameter Addition*	5,026 (0.5%)	$\exists p \notin P, p \in P' \land \neg \texttt{opt}(p)$	Breaking
6		Required Parameter Removal*	3,218 (0.3%)	$\exists p \in P, p \notin P' \land \neg \texttt{opt}(p)$	Breaking
7		Parameter Reordering*	840 (0.1%)	$\exists p_1, p_2 \in P \cap P', p_1 \prec p_2 \land p_1' \succ p_2'$	Breaking
8		Optional Parameter Addition	12,996 (1.2%)	$\exists p \notin P, p \in P' \land \texttt{opt}(p)$	Breaking
9		Optional Parameter Removal	4,063 (0.4%)	$\exists p \in P, p \notin P' \land \texttt{opt}(p)$	Breaking
10		Parameter Default Value Addition	1,006 (0.1%)	$\exists p \in P \cap P', \neg \texttt{opt}(p) \land \texttt{opt}(p')$	Breaking
11		Parameter Default Value Removal	851 (0.1%)	$\exists p \in P \cap P', \texttt{opt}(p) \land \neg \texttt{opt}(p')$	Breaking
12		Parameter Default Value Change	11,362 (1.0%)	$\exists p \in P \cap P', \texttt{def}(p) \neq \texttt{def}(p')$	Breaking
13	Field (f)	Field Removal*	182,826 (16.7%)	$f \in F \land f \notin F'$	Breaking
14		Field Addition*	190,619 (17.4%)	$f \notin F \land f \in F'$	Non-Breaking

"p" and "P" denote a single parameter and a parameter list, respectively. "c" and "C" denote a single class and a set of classes in a release, respectively ("m" and "M", "f" and "F" can be interpreted similarly). "*" means that this pattern also applies to Java. "'" denotes the new API version.
"$\texttt{opt}(p)$" means that the parameter p is optional. "$\texttt{def}(p)$" denotes the default value of p. "$a \prec b$" / "$a \succ b$" denotes parameter a is before/after b.

```python
def avg_pool(value, ksize, ...): # 1.13.1

def avg_pool(value, ksize, ..., input=None): # 1.14.0—rc0
# input = value if input is not specified

def avg_pool(input, ksize, ...): # 2.0.0—alpha0
```

Listing 1. Signature evolution of `tf.nn.avg_pool` in TensorFlow

(1) **Optional parameter addition and removal.** Compared to required parameters, the addition (33.0%) and removal (10.3%) of optional parameters are more frequent. Such changes can make API usages more flexible. For instance, in Flask 1.0, an optional parameter `static_host` was added to the constructor of the class `Flask` to allow users to specify the static resource hostname when creating the server object. We also observed cases where developers add or remove optional parameters in order to minimize the impact of breaking changes. Listing 1 shows a real example in TensorFlow, where developers intended to rename the parameter `value` of `tf.nn.avg_pool` to `input`. To minimize the risk of this change, they first added an optional alias `input` for `value` and delegated its value to the new alias. Then after a few releases, the `value` parameter was finally removed. **Parameter default value addition and removal.** Adding (2.6%) or removing (2.2%) default value will turn a required parameter into an optional one or the other way around. For example, in Django 2.0, the default value of parameter `on_delete` in the constructor of `django.db.models.ForeignKey` is removed to increase awareness about cascading model deletion. In Pandas 0.14.0, a default value `None` is added to the parameter `path_or_buf` in method `pd.DataFrame.to_csv` to create a default behavior.

(2) **Parameter default value change.** Changes of parameter default values account for a large proportion (28.9%) of parameter changes. We inspected the release logs and found that such changes are mainly intended for: (1) *con-*

sistency with external libraries or other APIs: for example, in TensorFlow 1.11.0, the default values of the constructor for `tf.keras.RandomUniform` are changed to match those in Keras; (2) *performance enhancement*: for example, the default value of the parameter `n_estimators` of `sklearn.ensemble.RandomForestClassifier` is changed from 10 to 100 in Scikit-learn 0.22 to enhance the ability of random forest classifier; and (3) *bug fixes*: for example, in Pandas 0.23.0, the default value of the parameter `convert_datetime64` is changed from `True` to `None` in `pandas.DataFrame.to_records` to fix a bug.

C. Comparison with Java

As a typical dynamic programming language, Python exhibits different API evolution patterns compared to Java as mentioned above. The most distinctive difference ensues from Python's flexible argument passing mechanism. In the following, we further compare the evolution of Python and Java frameworks from the perspective of breaking changes.

1) The rate of breaking changes: Based on our dataset, breaking changes happen much more frequently in our studied Python frameworks than in Java frameworks as reported by a recent study [8] (Figure 1). In addition, we observed that breaking changes in Python frameworks could occur in any stage of framework evolution (Figure 2).[5] The high rate of breaking changes could lead to frequent crashes of client applications, which we will discuss shortly in Section V-B. We further checked the release logs and commit messages of the six frameworks and found that developers often perform breaking changes to make their framework light-weight, consistent, or user-friendly. For example, in the release log of TensorFlow 2.0, developers mentioned *"many backward-incompatible API*

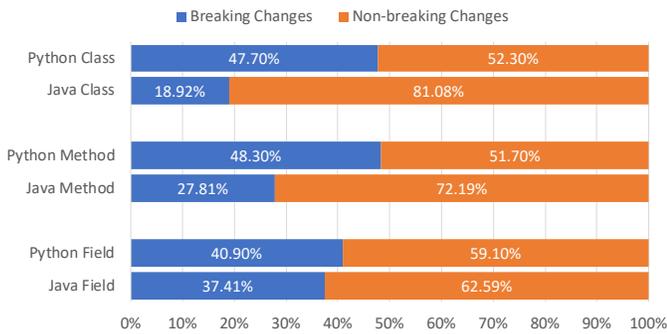[5]We only present the statistics of Pandas here due to page limit. More statistics can be found at https://doi.org/10.5281/zenodo.2756358.

Fig. 1. Comparison of the number of API changes in Python and Java

Fig. 2. # API changes in the evolution of Pandas

Framework	# Used Breaking APIs	# Affected Projects
TensorFlow	1,519 of 337,833 (0.4%)	626 of 1,000 (62.6%)
Pandas	1,085 of 23,543 (4.6%)	621 of 1,000 (62.1%)
Scikit-learn	218 of 12,500 (1.7%)	462 of 883 (52.3%)
Flask	19 of 202 (9.4%)	57 of 1,000 (5.7%)
Django	187 of 10,879 (1.7%)	629 of 1,000 (62.9%)
Keras	163 of 6,233 (2.6%)	688 of 1,000 (68.8%)
Total	3,191 of 391,190 (0.8%)	2,967 of 5,538 (53.6%)*

* Some project used multiple frameworks as explained in Section III-B

V. COMPATIBILITY ISSUES & FIXING STRATEGIES

A. Analysis Approach

1) Keyword formulation and search: To answer RQ2, we need to collect and analyze real compatibility issues induced by framework API evolution and the associated fixing patches in our selected 5,538 client applications. To ensure the quality of the collected data, we adopted an iterative process to formulate the search keywords. We started from keywords related to compatibility issues and API evolution based on our domain knowledge, which include *compatibility* and *version*. Then we refined the keyword set based on the quality of the resulting issues repeatedly until we obtained a sufficient number of confirmed issues in the search results. The final set of keywords include *compatibility*, *version*, *evolution*, *exception*, *TypeError*, *AttributeError*, and *ImportError*. We used these keywords, combined with the six names of our selected frameworks, to search both the issue tracking systems and commit histories of the 5,538 client applications. Note that searching commit histories is necessary since many issues could be fixed and committed without being documented in the issue reports [23].

2) Manual validation: Using the above search process, we collected 1,318 issue reports and 53,301 commits. We manually validated each issue report and 1,066 (2%) randomly sampled commits to find real compatibility issues induced by the API evolution of our studied Python frameworks. We didn't analyze all commits because the validation required extensive manual efforts and were very time-consuming. Finally, we obtained 409 issues from 124 issue reports and 219 commits.[7] The other issue reports and commits are not related to real compatibility issues but were retrieved because they accidentally contain our selected keywords. The manual validation process involved three authors. Two of them first independently checked each issue report or commit to validate if it is related to real issues of our concern. After independent checking, the two authors cross-validated their results. When disagreement happened, the third author would join the discussion and jointly make the final decision.

3) Characterizing issues: We then analyzed each of the 409 compatibility issues to understand their common symp-

---

changes have been made to clean up the APIs and make them more consistent." Developers also mentioned that the focus of TensorFlow 2.0 is *"simplicity and ease of use"*. Furthermore, due to the dynamic nature and flexibility of Python, framework developers may not easily notice or avoid breaking changes.

2) The impact of breaking changes: We also investigated the impact of breaking changes of the six frameworks on the 5,538 client applications following the methodology of the existing work [8]. Specifically, we analyzed how many APIs with breaking changes (breaking APIs for short) are used by the 5,538 projects. With static analysis, we found 3,191 such APIs.[6] The detailed breakdown is provided in Table III. We can see that the 3,191 APIs only account for a small percentage (0.8%) of all breaking APIs. This finding is expected because 96.4% of the breaking changes are API removals. Obviously, developers would not often use APIs that do not exist in a framework. We further analyzed how many of the 5,538 projects use the breaking APIs. To our surprise, we found 2,967 (53.6%) such projects. This indicates that the breaking APIs in our studied Python frameworks might have affected a wide range of client applications. Comparatively, such a ratio is only 2.5% in Java [8].

> **Answers to RQ1:** *We observed 14 API evolution patterns for Python frameworks, five of which are specific to Python due to its language features. We also found that comparing to Java, breaking API changes in Python frameworks happen more frequently and have more extensive impacts.*

---

[6]Static analysis provides a worse-case approximation of the impact.

[7]The number of issues we obtained is larger than 343 $(124+219)$ because developers fixed multiple issues in some commits.

toms and fixing strategies. To understand the symptoms, we investigated the following data associated with the issues: (1) issue reports, (2) commit messages, and (3) developers' comments in the corresponding code. We also tried to reproduce some issues when we could not understand their symptoms from the three data sources. To understand the fixing strategies, we primarily studied the patches of 347 issues. We did not study the remaining issues because they were not yet fixed at the time of this study or we failed to locate the corresponding fixing patches. This process also involved three authors. The methodology is similar to that of the above manual validation step. We present our findings below.

B. Symptoms of Compatibility Issues

We observed that the compatibility issues in client applications can lead to crashes or unexpected behaviors:

Application crashes (405/409). The vast majority of our studied issues led to unhandled exceptions, causing crashes of client applications. This ratio is surprisingly higher than that of the compatibility issues in other types of applications. For example, according to a recent study of the compatibility issues in Android applications, only 15 of the 67 compatibility issues induced by Android API evolution led to application crashes [5]. One possible explanation of such difference is that unlike SPL code, Python code is not strictly checked by a compiler before running. Therefore, severe errors such as crashes are deferred until runtime. These 405 crashing issues can be further classified into the following types of runtime exceptions, many of which are specific to Python.

(1) `AttributeError`: 48.1% (195/405) of the crashing issues are related to `AttributeError`. Such exceptions are raised when clients refer to an attribute that is not available, which is often caused by the removal or renaming/relocation of that API (see Table IV). Take issue #3 of the project *variational-autoencoder* [24] as an example. The application crashed because it referenced the function `kl` in the module `tf.contrib.distributions`, which was renamed to `kl_divergence` in TensorFlow 1.2.0.

(2) `TypeError`: 31.4% (127/405) of the crashing issues are related to `TypeError`. In Python, type checking is performed at runtime. A `TypeError` is raised when the arguments are incompatible with the parameter list. Such runtime exceptions are often caused by API parameter changes as shown in Table IV. For example, in the project *django-advanced-*

```
- tf.histogram_summary(var.op.name, var)
+ tf.summary.histogram(var.op.name, var)
```

Listing 2. Fixing an issue by replacing with new API calls

filters [25] (issue #72), developers encountered a `TypeError` when instantiating the Django class `AdvancedFilter`. The reason is that a required parameter `on_delete` was added to the constructor but the developers did not provide a corresponding argument. The parameter reordering of `tf.concat` in TensorFlow also caused a `TypeError` in the project *colornet* [26] (issue #12).

(3) `ImportError`: 14.1% (57/405) of the crashing issues are related to `ImportError`, which is raised when developers try to import a nonexistent module, function, or class. Here, we also count `ModuleNotFoundError` since it is a subtype of `ImportError`. For example, in *keras-contrib* [27] (issue #291), developers tried to import `normalize_data_format` from `keras.utils.conv_utils`, which was removed in Keras 2.2.1, and encountered `ImportError`. Such issues are often caused by API removal, renaming, and relocation.

(4) Other Errors: The remaining 6.4% (26/405) of the crashing issues are related to over 10 other types of runtime exceptions including `ValueError`, `KeyError`, and framework-specific exceptions, which are not commonly observed in our dataset. We do not further discuss them.

Unexpected behaviors (4/409). API evolution may not change an API's signature. For instance, in Section IV-B, we discussed that changes can be made on the default values of API parameters. Such subtle API changes may not crash client applications but could lead to unexpected behaviors, which is difficult to debug. For example, in Django 1.6.0, the default value of the parameter `default` in the constructor of the class `django.db.models.BooleanField` was changed from `False` to `None`. This affected the behavior of the application *django-dynamic-scraper* [28] (issue #41), which could run properly on older Django versions when the parameter `default` had the default value of `False`. To fix the issue, developers explicitly provided a `False` value to the `default` parameter when calling the API (commit `135b2ae`).

C. Fixing Strategies

By studying the patches of the 347 compatibility issues, we found that developers often adopt the following four types of strategies when fixing the issues. These four patterns cover only 267 patches. The remaining patches involve fixes that are too specific to be generalized.

Replacing with new API calls (182/347). The most common fixes are simply replacing the old API, which caused the compatibility issue, with the corresponding new API. Listing 2 shows an example that fixed an issue in the project *Colorization.tensorflow* [29] (commit `bec97c5`). The issue occurred because the API `tf.histogram_summary()` was relocated and renamed to `tf.summary.histogram()` during the evolution of TensorFlow. While such fixes are simple and straightforward, they do not guarantee an application's backward compatibility with older library versions.

```
- weight_decay = tf.mul(...)
+ try:
+     weight_decay = tf.multiply(...)
+ except AttributeError:
+     weight_decay = tf.mul(...)
```

Listing 3. Fixing an issue by using catching exceptions

Catching related exceptions (32/347). As discussed in Section V-B, API evolution may crash client applications with `ImportError` or `AttributeError`. To fix such issues, developers often catch such exceptions and fallback to alternative APIs. Listing 3 shows an example fix in the project *TensorMol* [30]. In the fixed version (commit `b8d07db`), the developers first tried to call `tf.multiply`, which is available in TensorFlow 1.0.0 and subsequent versions. If the API does not exist, then API `tf.mul`, which is available in the older versions of TensorFlow, will be invoked. An interesting finding is that developers tend to call the preferred APIs (usually those in newer versions) in the `try` clause and call the alternatives in the `except` clause. This may help avoid unnecessary computational overhead since exception handling is expensive. In addition, this strategy ensures backward compatibility, but it works only when an exception will be thrown, unexpected behaviors induced by API evolution cannot be solved by it.

Altering argument passing (29/347). As mentioned in section IV-B, parameter reordering would break the API calls with positional arguments. To fix such issues, developers may change from positional arguments to keyword arguments. The first part of Listing 4 demonstrates a fix in the project *tf_classification* [31] (commit `df29c8c`), where developers fixed a compatibility issue caused by the parameter reordering of `tf.concat()` using keyword arguments. On the other hand, parameter renaming would break the API calls with keyword arguments but would not affect the API calls with positional arguments. Hence, developers may choose to use positional arguments instead of keyword arguments to fix issues induced by parameter renaming. The second part of Listing 4 shows such a fix in the project *crfasrnn_keras* [32] (commit `6bfaee7`), where developers replaced keyword arguments with positional arguments when invoking `tf.nn.softmax()`. This strategy also ensures backward compatibility.

Checking library versions or API existence (24/347). Another commonly adopted strategy is to explicitly check the library versions before calling a target API. Listing 5 shows how Keras developer checked the TensorFlow version to decide appropriate API invocations (commit `1de4bf1`). Listing 6 shows how Keras developers checked whether the target API exists using `hasattr` before calling it (commit `aa18604`). Such explicit checking requires developers to be familiar with the evolution of the referenced library, such as knowing the exact library version in which the target API was changed. While this strategy ensures backward compatibility, it may not be suitable for inexperienced developers.

While different strategies can help fix different types of issues, we found that developers may also switch from one fixing strategy to another to fix similar issues. Take the project

```
# Positional arguments -> keyword arguments
- bboxes = tf.concat([...], 0)
+ bboxes = tf.concat(axis=[...], values=0)

# Keyword arguments -> positional arguments
- softmax_out = tf.nn.softmax(q_values, axis=0)
+ softmax_out = tf.nn.softmax(q_values, 0)
```

Listing 4. Fixing issues by altering argument passing

```
- v = tf.SparseTensor(..., shape=sparse_coo.shape)
+ if tf_major_version >= 1:
+     v = tf.SparseTensor(..., dense_shape=...)
+ else:
+     v = tf.SparseTensor(..., shape=...)
```

Listing 5. Fixing an issue by checking library versions

```
- tf.image_summary(weight.name, w_img)
+ if hasattr(tf, 'image_summary'):
+     tf.image_summary(weight.name, w_img)
+ else:
+     tf.summary.image(weight.name, w_img)
```

Listing 6. Fixing API relocation by using `hasattr`

Mask_RCNN [33] as an example. When the compatibility issue #566 was reported, developers replaced the call to an old API with a call to the new one to fix the issue. We have discussed that such a simple fix cannot guarantee backward compatibility. Later on, a similar issue #669 was reported, in which backward compatibility is an important concern. As a result, developers caught `ImportError` in commit `9e0997a` to fix this issue. Another example is in the project *tensorlayer* [34] (commit `989c584`), where developers changed from checking library versions to catching related exceptions.

> **Answers to RQ2:** *Python framework API evolution may cause crashes or unexcepted behaviors in client applications, including over 10 types of runtime exceptions, many of which are specific to Python. Developers mainly adopt four types of strategies to fix compatibility issues caused by framework API evolution, three of which guarantee backward compatibility while one does not.*

VI. PYCOMPAT : A COMPATIBILITY ISSUE DETECTOR

Our findings in Section V-B suggest that the evolution of Python framework APIs often leads to crashes, which severely diminishes the reliability and usability of client applications. Yet, it is often tedious and time-consuming for developers to test their applications in different environments to guarantee compatibility. This motivates us to design and implement a tool, PYCOMPAT, based on our empirical findings, to help Python developers automatically identify compatibility issues in the early development phase of their projects. The API evolution patterns summarized in Table II are atomic changes, which are composed of a single operation. We focus on atomic changes in our empirical study since: 1) the detection of such changes can be fully automated, and thus the detection approach can be applied to a large number of API changes; 2) existing studies also focus on understanding API evolution patterns in terms of atomic changes [8], and thus we follow such a practice to facilitate fair comparisons with them (see
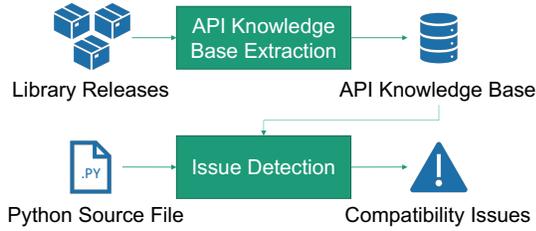
Fig. 3. Workflow of PYCOMPAT

Section IV-C). On the contrary, our tool is designed to detect high-level changes, which are composed of multiple atomic changes, due to two reasons. First, detecting atomic changes will generate an overwhelming number of warnings for those changes, which might not be useful to developers. Second, detecting high-level changes can generate more succinct and actionable reports. The following subsections present the details of PYCOMPAT and a preliminary evaluation of it.

A. Tool Design

PYCOMPAT aims to detect compatibility issues caused by common types of breaking API changes via static analysis. Specifically, we designed it to detect issues caused by API renaming/relocating (**ARR**) and parameter renaming (**PRN**), which are high-level changes composed of atomic ones such as addition and removal operations. Currently, PYCOMPAT does not support detecting issues caused by other high-level API changes such as parameter reordering due to (1) the lack of precise type information in static analysis, and (2) the low frequency of such changes observed in our dataset.

PYCOMPAT works in two phases: *API Knowledge Base Extraction* and *Issue Detection* (Figure 3). The first phase analyzes the releases of a framework and synthesizes a knowledge base that models the high-level changes of framework APIs. The second phase takes the API knowledge base as input and performs static analysis on a given Python source file to locate potential compatibility issues via light-weight rule checking.

1) API Knowledge Base Extraction: Given a set of library releases, PYCOMPAT first extracts the APIs defined in each library release as described in Section IV-A, and classifies API changes into patterns according to the rules in Table II. With the extracted APIs and evolution patterns, we then construct a knowledge base KB to model high-level API changes including **ARR** and **PRN**. Note that unlike analyzing atomic changes, identifying the two types of high-level changes cannot be easily automated. We need to manually inspect the changelogs of a target framework to precisely locate renamed methods or parameters. Finally, each entry in KB encodes the detailed information of an evolved API, including (1) the names and parameter lists of the API before and after the evolution, (2) the mappings between renamed parameters, and (3) the release version in which the API evolved.

2) Issue Detection: The detection algorithm takes two inputs: (1) KB, the API knowledge base built in the first phase of PYCOMPAT, and (2) AST, the Abstract Syntax Tree of the Python source file under analysis. Firstly, we walk through

the AST to obtain the call sites of the APIs defined in the framework. Then, we analyze each call site to check if it uses an evolved API by matching the following rules:

ARR Checking Rule: This rule helps find compatibility issues caused by API renaming/relocation. An ARR rule is matched if there is an entry r in KB such that (1) r describes the renaming/relocation of an API, and (2) the API name of the call site matches the API name in r before or after evolution.

PRN Checking Rule: This rule helps find compatibility issues caused by parameter renaming. An PRN rule is matched if there is an entry r in KB such that (1) r describes the parameter renaming of an API, (2) the API name of the call site matches the API name in r,[8] and (3) there are keyword arguments passed to renamed parameters encoded in r.

When there is a rule matched at a call site, we further check if developers have already taken actions to avoid compatibility issues. Specifically, we perform backward slicing at the call site and analyze the obtained slice to check if there exist any potential fixes. We conservatively consider that a fix exists if any of these conditions is satisfied: (1) the call site is wrapped by a `try...except` statement, (2) there exist statements in the slice that check the framework versions, and (3) there exist statements in the slice that check API existence. If there is no sign of potential fixes, a warning will be reported.

B. Preliminary Evaluation

In this section, we evaluate PYCOMPAT with real-world projects that use TensorFlow. We focus on TensorFlow applications because it is one of the most popular Python frameworks. It has tens of releases and a large number of client applications.

For the evaluation, we manually prepared an API knowledge base containing 45 entries that encode the evolution details of TensorFlow APIs. We randomly selected 10 client projects of TensorFlow on GitHub which has: (1) more than 50 stars, (2) commits, active issues or pull requests in the last six months, and (3) no explicit requirement of TensorFlow versions in its description page. These constraints help select popular and active projects that are likely to have compatibility issues. Table V lists our selected projects.

We ran PYCOMPAT on the latest revision of the 10 projects. After performing compatibility analysis at 7,000+ call sites of TensorFlow APIs in these projects (Column 4 of Table V), PYCOMPAT raised 156 and 105 warnings induced by **ARR** and **PRN**, respectively. We then manually checked all the 261 warnings and classified them into true positives (TP) and false positives (FP). Columns 5-8 of Table V report the results.

In total, 231 detected warnings are true positives (precision is 88.5%). The 30 false positives all come from one project. We checked its code and found that developers created two directories to maintain two versions of the project, one for old TensorFlow APIs and the other for new TensorFlow APIs. That means they are aware that TensorFlow API evolution could incur compatibility issues in their project and have already prevented such issues, although using an unusual strategy. Hence, our detected issues are not useful to them.

---

[8]In our definition, API names remain unchanged in parameter renaming.

88

TABLE V
EXPERIMENTAL SUBJECTS AND CHECKING RESULTS

Repository Name	Revision	# Stars	# API Call Sites	# ARR	# PRN	# TP	# FP	Issue ID(s)
ck-tensorflow [35]	c1ff67d	77	446	3	0	3	0	114 (Fixed)
keras [36]	9d33a02	40,945	612	1	1	2	0	12710 (Ignored)
tensorflow-yolov3 [37]	970cd22	783	340	7	8	15	0	117 (Fixed), 136 (Pending)
tensorflow-101 [38]	aa80965	1,041	2,650	42	21	63	0	19 (Fixed), 21 (Fixed)
tensorflow-playground [39]	349db97	71	272	12	0	12	0	8 (Pending)
TF-and-DL-Tutorial [40]	1b70523	2,234	798	31	0	1	30	38 (Fixed)
Valified_Code_Classify [41]	9043f4a	77	69	1	0	1	0	3 (Pending)
spotify-tensorflow [42]	5f607ad	79	129	0	1	1	0	193 (Confirmed)
R2CNN_FPN_Tensorflow [43]	0dc1b20	317	2,097	59	71	130	0	66 (Ignored), 68 (Ignored)
tensorflow-triplet-loss [44]	fc69836	522	180	0	3	3	0	41 (Confirmed)
Total			7,593	156	105	231	30	13

We further reported the 231 true warnings to the developers of the corresponding projects. We grouped similar warnings into 13 issue reports in order not to overwhelm developers. The issue report IDs are provided in the last column of Table V. We also proposed patches in some reports to help developers fix the issues. At the time of paper submission, developers have replied to 10 of the 13 issue reports. The 78 issues mentioned in 7 reports have been confirmed. Moreover, developers quickly fixed 74 issues mentioned in 5 reports within one week after acknowledging the issues. There are 3 reports containing issues that developers did not want to confirm or fix. We checked their comments on the issue reports and found that it is mainly because these developers do not want their projects to be compatible with older versions of libraries. For example, Keras developers made this comment on our issue report: *"Keras generally only supports TF versions going back 2 releases. We're currently on 1.13, so we should only support TF down to 1.11. If you need to use an older version of TensorFlow, you should downgrade Keras as well"*.

Based on the above findings, we can see that PY-COMPAT can find compatibility issues that are helpful to developers. This demonstrates the usefulness of our work.

VII. DISCUSSIONS

A. Threats to Validity

1) Subject Selection: Our selected Python frameworks and client applications might not be representative. To mitigate the threat, we selected frameworks from three different domains. These frameworks are also widely used by developers. For client applications, we selected a large number of diverse projects (5,538 in total) built on the selected frameworks.

2) APIs Change Pattern Classification: Due to the fact that Python is dynamically typed, we cannot get the type information without running the programs. Therefore, our classification does not include changes that are related to type, such as the type change of the default value for parameters.

3) Issue Selection: We detected compatibility issues using self-defined keywords, which may limit the diversity and representativeness of the collected issues. To mitigate the threat, we adopted a process to iteratively discover and refine keywords based on the quality of the retrieved data. It is possible that some Python-specific keywords (e.g., TypeError) in our keyword set would lead to a biased dataset (e.g., many of the collected issues are caused by TypeError). However, we observed that our selected keyword set also helped collect many compatibility issues caused by other types of errors (e.g.,ValueError, KeyError, etc.), which demonstrates the generality of our collected dataset.

4) Manual Analyses: Our empirical study and evaluation involved much manual effort, which might be subjective and biased. To reduce the threat, multiple authors were involved to perform independent checking and cross-validated all results. We also released our dataset for public access and scrutiny.

B. Observations & Implications on Future Work

1) API Evolution in Different Phases: We observed that in different phases, library API evolution may exhibit different patterns: (1) APIs evolve more frequently between a stable version and its subsequent unstable version, and (2) deprecated APIs are usually removed in major version releases. Research efforts can be spent on studying API evolution patterns in different phases of libraries.

2) Automated Modeling of API Evolution: In this work, we built the API evolution knowledge base when implementing PYCOMPAT in a semi-automated way. It is interesting to study how to automatically model the high-level changes of Python framework API evolution. There are two outstanding challenges. First, since Python is a DPL, there is little type information in the code although type annotations can be used. It is difficult to map parameters of evolved APIs without knowing their types. Second, Python allows arbitrary keyword arguments (i.e., written in **kwargs), which could aggregate multiple arguments into a dictionary. Such arguments can only be retrieved in function bodies. It is difficult to detect the parameter list of such functions without analyzing library code.

3) Automated Patching of Compatibility Issues: Currently, our work only focuses on detecting compatibility issues. With our observed fixing strategies, it is promising to design techniques to automatically generate patches to fix compatibility issues based on the API evolution knowledge base.

4) API Evolution of Other DPL Frameworks: The frequent evolution of DPL frameworks could induce various compatibility issues in client applications. In this work, we performed an exploratory study on Python frameworks. Future work can study frameworks in other DPLs such as JavaScript and Ruby.

VIII. RELATED WORK

A. API Evolution of Libraries

1) Understanding API Evolution & Its Impact: Des Rivières [45] provided a catalog of Java API changes and discussed how to evolve APIs while maintaining compatibility with client code. Meng et al. [46] proposed a history-based matching approach to identifying and understanding the API evolution of Java frameworks. Dig et al. [15] classified API evolution in Java libraries from the perspective of refactoring. They found that 80% of the breaking changes are API refactoring. Xavier et al. [8] investigated the API changes in 317 real-world Java libraries and observed a high rate of breaking changes (27.99%), which impacted 2.54% of their clients. Dietrich et al. [4] analyzed binary compatibility problems in OSGi-based systems. Tao et al. [47] found that alternative API usages could improve runtime performance and proposed a novel approach to exploring implementations for the same task using different APIs. Many studies also focused on specific domains or ecosystems. For example, Wu et al. [48] analyzed the frequency of API evolution patterns in Apache and Eclipse ecosystem and evaluated the impact of the API evolution. Li et al. [49] investigate web service API evolution and how it affects client applications. McDonnell et al. [50] and Wei et al. [5] studied the API stability and evolution in Android frameworks and their induced issues. Most recently, Zhang et al. [51] found that API evolution might be the root cause of various bugs in deep learning programs. In comparison, our work is the first to systematically study the API evolution of dynamic language frameworks. Although we focused on Python, our findings may be useful to future studies on the API evolution of frameworks in other DPLs.

2) How Developers React to API Evolution: There are also studies on how developers deal with API evolution problems. For instance, Hora et al. [52] studied the API evolution problems in the Pharo ecosystem and found that developers often do not quickly react to such problems (median reaction time is 34 days). Wei et al. [53] found that Android developers often check API levels before API invocations to avoid compatibility issues. Similar to [53], we observed that Python developers also check library versions when using APIs to avoid compatibility issues. However, we also observed other strategies, such as altering parameter passing, to deal with compatibility issues, which are specific to Python applications.

B. API Evolution Supporting Techniques

1) API Misuses Detection: Researchers have proposed various techniques to tame issues caused by misusing APIs. For example, FicFinder [5] and CiD [54] can help detect potential compatibility issues induced by misusing platform-specific or frequently evolved APIs on Android. MUTAPI [55] discovers API misuses patterns via mutation analysis. In our work, we also proposed PYCOMPAT, a tool to detect compatibility issues by searching usages of evolved APIs. Although the goal of the tools is similar, PYCOMPAT are specifically designed to handle different issues for Python applications.

2) API Migration Support: API migration is important but challenging. Several studies proposed techniques or transformation languages to automate API migration [56]. For example, Henkel et al. [57] proposed an Eclipse plugin called CatchUp! to record how developers refactor API usages and then replay the refactoring on other client code to ease API migration. Muller et al. [58] proposed a framework, Coccinelle, to automate required code modifications (semantic patches) to support the collateral evolution of Linux libraries and their dependent service-specific code. SemDiff [59] recommends replacements for framework methods that are accessed by a Java program but deleted during evolution. More recently, Wang et al. [60] proposed a declarative language, PATL, to support transforming programs between different versions of the same API. Comparing to our work, the existing work mostly focused on the ability to match multiple related API calls but do not build a knowledge base for API evolution. Furthermore, the proposed language and frameworks rely on the static type system of the target language, and cannot be easily applied to DPLs such as Python.

IX. CONCLUSION

In this paper, we performed an empirical study to understand the API evolution in Python frameworks. Via analyzing the API changes in 288 releases of six popular Python frameworks, we found 14 common API evolution patterns in Python frameworks, including five patterns that are not observed in Java frameworks. We also investigated 409 compatibility issues in Python applications, which were induced by API evolution of the six frameworks. We found common symptoms and causes of the issues and observed four strategies that developers often use to fix compatibility issues. Based on our empirical study, we designed and implemented PYCOMPAT, a tool that can automatically detect compatibility issues in Python applications. Experiments on 10 real-world projects show that PYCOMPAT can effectively detect compatibility issues that are of concern to developers.

In our future work, we plan to enhance the API change extraction algorithm (e.g., recognizing and unifying API aliases caused by mapping) and study more Python frameworks to characterize their API evolution. We also plan to automate the API knowledge base extraction process of PYCOMPAT to make our technique more practical.

REFERENCES

[1] "Tiobe index for october 2019." [Online]. Available: https://www.tiobe.com/tiobe-index/

[2] "Pypl popularity of programming language." [Online]. Available: http://pypl.github.io/PYPL.html

[3] "The incredible growth of python." [Online]. Available: https://stackoverflow.blog/2017/09/06/incredible-growth-python/

[4] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, 2014, pp. 64–73. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[5] L. Wei, Y. Liu, and S. Cheung, "Taming android fragmentation: characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 226–237. [Online]. Available: https://doi.org/10.1145/2970276.2970312

[6] "Stack overflow." [Online]. Available: https://stackoverflow.com

[7] "Github - the world's leading software development platform." [Online]. Available: https://github.com

[8] L. Xavier, A. Brito, A. C. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, 2017, pp. 138–147. [Online]. Available: https://doi.org/10.1109/SANER.2017.7884616

[9] "Dynamic programming language (wikipedia)." [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_programming_language

[10] "Python documentation." [Online]. Available: https://docs.python.org/3/tutorial/controlflow.html#default-argument-values

[11] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 319–330. [Online]. Available: https://doi.org/10.1145/3236024.3236056

[12] "pypa/virtualenv: Virtual python environment builder." [Online]. Available: https://github.com/pypa/virtualenv

[13] "pip - the python package installer." [Online]. Available: https://pip.pypa.io

[14] "Pypi - the python package index." [Online]. Available: https://pypi.org/

[15] D. Dig and R. E. Johnson, "How do apis evolve? A story of refactoring," *Journal of Software Maintenance*, vol. 18, no. 2, pp. 83–107, 2006. [Online]. Available: https://doi.org/10.1002/smr.328

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[17] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.

[20] "pallets/flask: The python micro framework for building web applications." [Online]. Available: https://github.com/pallets/flask

[21] "django/django: The web framework for perfectionists with deadlines." [Online]. Available: https://github.com/django/django

[22] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[23] Y. Liu, J. Wang, L. Wei, C. Xu, S.-C. Cheung, T. Wu, J. Yan, and J. Zhang, "DroidLeaks: A Comprehensive Database of Resource Leaks in Android Apps," *Empirical Software Engineering*, pp. 325–334, 2019.

[24] "altosaar/variational-autoencoder: Variational autoencoder implemented in tensorflow and pytorch." [Online]. Available: https://github.com/altosaar/variational-autoencoder

[25] "modlinltd/django-advanced-filters: Add advanced filtering abilities to django admin." [Online]. Available: https://github.com/modlinltd/django-advanced-filters

[26] "pavelgonchar/colornet: Neural network to colorize grayscale images." [Online]. Available: https://github.com/pavelgonchar/colornet

[27] "keras-team/keras-contrib: Keras community contributions." [Online]. Available: https://github.com/keras-team/keras-contrib

[28] "holgerd77/django-dynamic-scraper: Creating scrapy scrapers via the django admin interface." [Online]. Available: https://github.com/holgerd77/django-dynamic-scraper

[29] "shekkizh/colorization.tensorflow: Image colorization using cnns in tensorflow." [Online]. Available: https://github.com/shekkizh/Colorization.tensorflow

[30] "jparkhill/tensormol: Tensorflow + molecules = tensormol." [Online]. Available: https://github.com/jparkhill/TensorMol

[31] "visipedia/tf_classification: Training, evaluation and testing code for image classification using tensorflow." [Online]. Available: https://github.com/visipedia/tf_classification

[32] "sadeepj/crfasrnn_keras: Crf-rnn keras/tensorflow version." [Online]. Available: https://github.com/sadeepj/crfasrnn_keras

[33] "matterport/mask_rcnn: Mask r-cnn for object detection and instance segmentation on keras and tensorflow." [Online]. Available: https://github.com/matterport/Mask_RCNN

[34] "tensorlayer/tensorlayer: Deep learning and reinforcement learning library for scientists." [Online]. Available: https://github.com/tensorlayer/tensorlayer

[35] "ctuning/ck-tensorflow: Collective knowledge components for tensorflow." [Online]. Available: https://github.com/ctuning/ck-tensorflow

[36] "keras-team/keras: Deep learning for humans." [Online]. Available: https://github.com/keras-team/keras

[37] "Yunyang1994/tensorflow-yolov3: pure tensorflow implement of yolov3 with support to train your own dataset." [Online]. Available: https://github.com/YunYang1994/tensorflow-yolov3

[38] "burness/tensorflow-101: learn code with tensorflow." [Online]. Available: https://github.com/burness/tensorflow-101

[39] "wangz10/tensorflow-playground: Implementations of some deep learning models using tensorflow with scikit-learn like apis." [Online]. Available: https://github.com/wangz10/tensorflow-playground

[40] "Creatcodebuild/tensorflow-and-deeplearning-tutorial: A tensorflow & deep learning online course i taught in 2016." [Online]. Available: https://github.com/CreatCodeBuild/TensorFlow-and-DeepLearning-Tutorial

[41] "Symphonypy/valified_code_classify: Python3/tensorflow." [Online]. Available: https://github.com/SymphonyPy/Valified_Code_Classify

[42] "spotify/spotify-tensorflow: Provides spotify-specific tensorflow helpers." [Online]. Available: https://github.com/spotify/spotify-tensorflow

[43] "yangxue0827/r2cnn_fpn_tensorflow: R2cnn: Rotational region cnn based on fpn (tensorflow)." [Online]. Available: https://github.com/yangxue0827/R2CNN_FPN_Tensorflow

[44] "omoindrot/tensorflow-triplet-loss: Implementation of triplet loss in tensorflow." [Online]. Available: https://github.com/omoindrot/tensorflow-triplet-loss

[45] J. d. Rivières, "Evolving java-based apis," Oct 2007. [Online]. Available: http://wiki.eclipse.org/Evolving_Java-based_APIs

[46] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 353–363. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337265

[47] Y. L. Z. X. S. Q. Yida Tao, Shan Tang, "How do api selections affect the runtime performance of data analytics tasks?" in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19, 2019.

[48] W. Wu, F. Khomh, B. Adams, Y. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2366–2412, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9411-7

[49] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service api evolution affect clients?" in *2013 IEEE 20th International Conference on Web Services*, June 2013, pp. 300–307.

[50] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, 2013, pp. 70–79. [Online]. Available: https://doi.org/10.1109/ICSM.2013.18

[51] Y. Zhang, Y. Chen, S. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 129–140. [Online]. Available: https://doi.org/10.1145/3213846.3213866

[52] A. C. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to API evolution? the pharo ecosystem case," in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, 2015, pp. 251–260. [Online]. Available: https://doi.org/10.1109/ICSM.2015.7332471

[53] L. Wei, Y. Liu, S.-C. Cheung, H. Huang, X. Lu, and X. Liu, "Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps," *IEEE Transactions on Software Engineering*, 2018.

[54] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 153–163. [Online]. Available: https://doi.org/10.1145/3213846.3213857

[55] M. Wen, Y. Liu, R. Wu, X. Xie, S. Cheung, and Z. Su, "Exposing library API misuses via mutation analysis," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019,* *Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 866–877. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00093

[56] M. Lamothe and W. Shang, "Exploring the use of automated API migrating techniques in practice: an experience report on android," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, 2018, pp. 503–514. [Online]. Available: https://doi.org/10.1145/3196398.3196420

[57] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support API evolution," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 274–283. [Online]. Available: https://doi.org/10.1145/1062455.1062512

[58] G. Muller, Y. Padioleau, J. L. Lawall, and R. R. Hansen, "Semantic patches considered helpful," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 90–92, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151374.1151392

[59] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for API evolution," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 599–602. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070565

[60] C. Wang, J. Jiang, J. Li, Y. Xiong, X. Luo, L. Zhang, and Z. Hu, "Transforming Programs between APIs with Many-to-Many Mappings," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 25:1–25:26. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6119