Understanding Type Hints in Python Libraries and Frameworks: Early Insights

Thiago Roberto Magalhães
trm@ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

João Eduardo Montandon
joao@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

Abstract

In Python, type hints allow developers to annotate variables and functions with explicit type information, improving code clarity and reliability. This paper presents an initial study on the adoption and usage of type hints in Python libraries and frameworks. By analyzing 1,000 popular GitHub repositories, we address two questions: (a) whether libraries and frameworks adopt type hints, and (b) how type hints are used in these components. While 91% of libraries use type hints at least once, this adoption is not consistent, as half of them cover only 13.6% of their members with types. Considering libraries with systematic usage, maintainers prioritize annotating function parameters and return types (45.8% and 35.9% median coverage), mainly using built-in types (73.0%). These findings highlight the role of type hints in APIs maintenance while pointing to opportunities for improved tooling and automation.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; **Software development techniques**; **Language types**.

ACM Reference Format:
Thiago Roberto Magalhães and João Eduardo Montandon. 2026. Understanding Type Hints in Python Libraries and Frameworks: Early Insights. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3794763.3794792

1 Introduction

Libraries and frameworks are the building blocks of modern software development [10, 14, 23]. These components provide pre-built functionalities that speed up the implementation of software products by allowing developers to focus on core business logic. This modular approach not only enhances code reusability but also reduces maintenance costs, allowing teams to deliver robust and scalable solutions faster and cheaper. In this context, the Python ecosystem emerges as one of the most active [22, 24], offering solutions for a wide range of software applications, from web development to data science and machine learning [15, 25, 28].

Python was designed to be simple and clean, making it easy to read and write code [17, 19]. The language is dynamically typed, i.e., variable types are determined and checked at runtime rather than at compile time [27]. Such features give developers a large degree of freedom when writing code, since they can handle different objects uniquely based on their behavior.[1] In practice, developers can implement programs faster and with less boilerplate [13, 27]. However, dynamic typing can also lead to potential issues, such as runtime type errors that are only detected during execution [1, 3, 4, 6, 27]. To aid static analysis tools in detecting type inconsistencies before runtime, Python introduced type hints in 2014 at version 3.5 [17]. This feature allows developers to annotate variables and functions with explicit type information, thus providing more context about which types are expected when using these annotated members.

Type hints have gained significant traction within the Python community since their introduction [24], being adopted in popular projects such *NumPy*[2] and *Pandas*.[3] Prior work in the literature has shown the benefits of using types to improve error detection and software maintainability [1, 3, 5–7, 9, 11–13]. Yet, little is known about how type hints are adopted in libraries and frameworks.

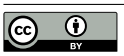
Proposed Study. We present a first outlook on the use of type hints in real-world Python libraries and frameworks. We collected and extracted type annotation data from 1,000 popular Python repositories hosted on GitHub, identified which ones are libraries and frameworks, and analyzed their type hint usage. Specifically, we investigated the following research questions:

- *RQ1. Do Libraries and Frameworks Adopt Type Hints?* Despite 91% of libraries have used type hints at least once, this adoption is not employed consistently across their codebase. In fact, half of the libraries covered only 13.6% of their members with types.
- *RQ2. How Type Hints are Used in Libraries and Frameworks?* Considering libraries with systematic type hint usage, maintainers focus on annotating function parameters and return types, with median coverage of 45.8% and 35.9%, respectively. Furthermore, built-in types are preferred, accounting for 73.0% of all type annotations in half of the libraries.

We believe the emerging insights from this study shed light on how type hints are being used in Python libraries and frameworks, guiding future research in this topic.

2 Understanding Type Hints

Type Hints are one of the biggest changes in Python history [17]. Introduced in PEP 484[4] in 2014, this proposal specifies the syntax and semantics for explicit type declarations in function arguments,



ICPC '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2482-4/2026/04
https://doi.org/10.1145/3794763.3794792

[1] https://en.wikipedia.org/wiki/Duck_typing
[2] https://numpy.org/
[3] https://pandas.pydata.org/
[4] https://peps.python.org/pep-0484/

return values, and variables. Type hints are provided as a *gradual type system*, where type annotations (a) are optional, since Python checker should not emit warning for code without types; (b) do not prevent inconsistent values from being assigned during runtime, and (c) do not improve programs performance since types are not added to the program's bytecode. *The goal is to provide more type information to static analysis tools so they can detect inconsistencies more effectively before runtime.*

A Type Hint Example. Figure 1 presents an example about the use of type hints. Lines 1 and 2 define the `concat` function, which receives two strings as parameters—`first` and `second`—and prints the concatenation of both on the screen. Note that both parameters are annotated with the type `str`, the string type in Python, and the return value is annotated with `None`, representing a method without return value.

```
1 def concat(first: str, second: str) -> None:
2     print(first + second)
3
4 concat("Type", "Hint")
5 concat(1, 2)
```

Figure 1: A function with type hints in Python.

Line 4 calls `concat` with `"Type"` and `"Hint"` as parameters, which prints `"TypeHint"`. This call is not only valid but also expected, since the type of the values involved in the call match the types annotated in `concat`'s definition. Static analysis tools emit no warning on this call.

On the other hand, line 5 invokes the function with `1` and `2` as parameters. This call clearly violates the annotated types, since both values belong to the `int` type. Any static analysis tool shall emit a warning on this line, showing that the types are incompatible. Nonetheless, this code will still execute and print `3` as a result, i.e., it will sum both `int` values. This happens because, in a gradual type system, type hints do not prevent runtime type errors; they are intended to aid static analysis and tooling.

3 Study Design

We advocate the usage of type hints is particularly important in *libraries & frameworks*, where the lack of clarity, correctness, and the presence of errors impact the reliability of the features provided by them [4, 8, 12, 15]. This study provides an initial assessment of how type hints are being used in Python third-party components.

3.1 Resarch Questions

In this initial study, we elaborated two research questions to be answered.

RQ1. Do Libraries and Frameworks Adopt Type Hints? Prior work shows that the adoption of type hints is becoming more popular on Python projects [3, 13]. However, we lack empirical evidence on how this feature is being adopted by *libraries & frameworks*. Thus, we first verify whether projects representing third-party components do use type hints in their codebase.

RQ2. How Type Hints are Used in Libraries and Frameworks? We conduct an exploratory investigation to understand how maintainers utilize type hints in *libraries & frameworks*. Particularly, we analyzed *where* type hints are mostly employed, and *what* are the most frequent annotated types.

3.2 Data Collection

The study design and methodology adopted in this work is depicted at Figure 2, and is described as follows.

1. Fetch GitHub Repositories. We initially gathered a collection of 1,000 Python repositories using the GitHub API, ranked by the number of stars. This approach ensured our dataset to contains well-established and widely adopted projects in the Python ecosystem from a diverse range of domains such as web development, data science, and machine learning. We recorded repositories metadata—such as repository name, number of stars, and number of Python files—for later reference.

2. Repository Selection. To improve data quality and reduce noise, we applied several filtering criteria. First, repositories containing fewer than 30 Python source files were excluded, since small projects rarely contain sufficient code to provide meaningful insights into annotation practices. Second, we removed repositories primarily designed for educational purposes (e.g., tutorials, course materials, or example snippets), as they tend to exhibit artificially simplified code [2, 18]. Third, we excluded repositories that were forks or duplicates of other projects in our dataset to avoid redundancy and ensure diversity in our analysis. Finally, we disconsidered those whose primary language was not English. After this step, 720 repositories were selected.

3. Repository Classification. One author manually analyzed the name, description, and tags of each repository to classify them as either *libraries & frameworks* or *others* [2]. The second author independently categorized a sample of 100 repositories to assess inter-rater agreement. The resulting Cohen's Kappa was 0.52, indicating moderate agreement between both classifications [26]. Differencies between both classifications were discussed to reach a consensus among the raters. Each repository classified as *libraries & frameworks* was cloned locally to enable further static analysis of its source code.

4. Type Hints Extraction. We performed an in-depth static analysis to identify and extract the type hints implemented in their codebase. For this, we built a custom analyzer in Python based on the Abstract Syntax Tree (AST) module.[5] For each repository, we parsed every python file, generated their *ast*, and visited the following nodes to extract their type hints.

- *Annotated assignments* (`AnnAssign`), to detect type hints applied to variables, regardless of their declaration scope.
- *Function definitions* (`FunctionDef`), to extract both parameter and return type annotations.

5. Type Hints Classification. We classify the type hints into one of the following three categories according to their origin.

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

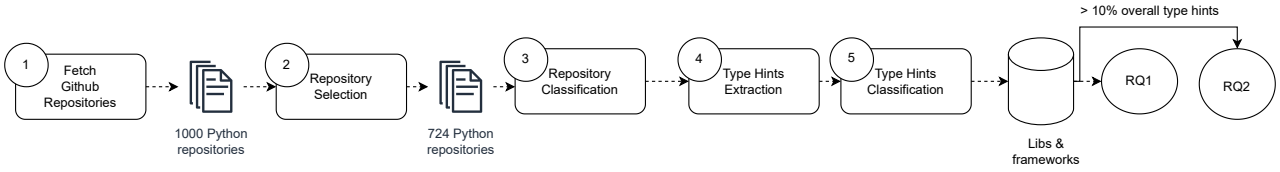


Figure 2: Data Collection adopted in this study.

- *Built-in types*, representing types that are provided by the Python language, such as `int`, `str`, and `dict`. These types were mapped based on the list of built-in types in the official Python documentation.[6]
- *Local types*, corresponding to types declared locally in the repository, such as classes, data structures, or aliases. To make this verification possible, we leveraged all type declarations performed in the project where the annotated type was found and verified whether the annotated type matched any of these locally declared classes.
- *External types*, referring to types that do not belong to the previous categories, thus originating from third-party libraries like `Tensor` from *pytorch*, and `DataFrame` from *pandas*.

Before applying this classification, we performed a normalization step to ensure consistent comparison across all extracted types. We (a) removed any package prefixes, e.g., converting `typing.Dict` to `Dict`, and (b) unified equivalent constructs, e.g., making `Dict` ≡ `dict`. To avoid excessive fragmentation, we treated complex annotations conservatively: for generics, only the container type was considered; for union and optional types, we decomposed them into their base components, excluding `None`.

3.3 The Resulting Dataset

From the 720 repositories selected for our study, 152 (21%) are *libraries & frameworks*. For each type hint detected, we stored: (a) the file path, (b) the qualified member name, (c) the annotated type, (d) the member category (local variable, parameter, or return), and (e) a contextual code snippet with the extracted annotation. In total, this procedure analyzed 4,670,698 source code members and extracted 649,099 type hints. We used this baseline to answer the research questions proposed for this work.

Type Hint Coverage Metric. To properly assess the adoption of type hints across *libraries & frameworks*, we computed a proportion-based metric called *type hint coverage*. For a given repository, this metric calculates the ratio of annotated members to the number of eligible members in the context under analysis. For example, the *flask* framework contains 4,614 overall members—i.e., parameters, return commands, and variables—of which 1,011 are annotated with type hints. Thus, its *overall type hint coverage* is 21.9% $(1,011/4,614)$. From these, 1,808 are parameters and 524 of them are annotated, resulting in a *parameter type hint coverage* of 28.9% $(524/1,808)$. We used this metric as it allows fair comparison between repositories of different sizes and characteristics.

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

4 Results

RQ1. Do Libraries and Frameworks Adopt Type Hints? Overall, 139 out of 152 (91%) *libraries & frameworks* contain at least one type hint in their codebase. The project with the highest type hint coverage is *openai-python*—the official library for the OpenAI API—with 61%, follwed by *altair*—library for data visualization—with 53%. Other popular projects show consistent adoption across their codebases. *Pytest*—one of the most popular Python libraries for software testing—annotates 39% of its members. *FastAPI*—a well-known web development framework—fulfilled 35% of its members with type hints.

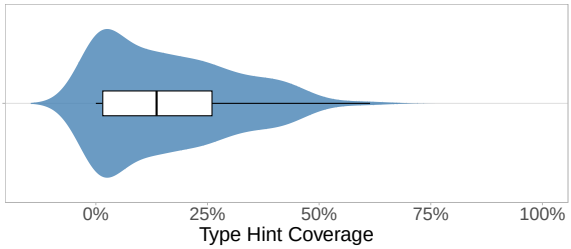


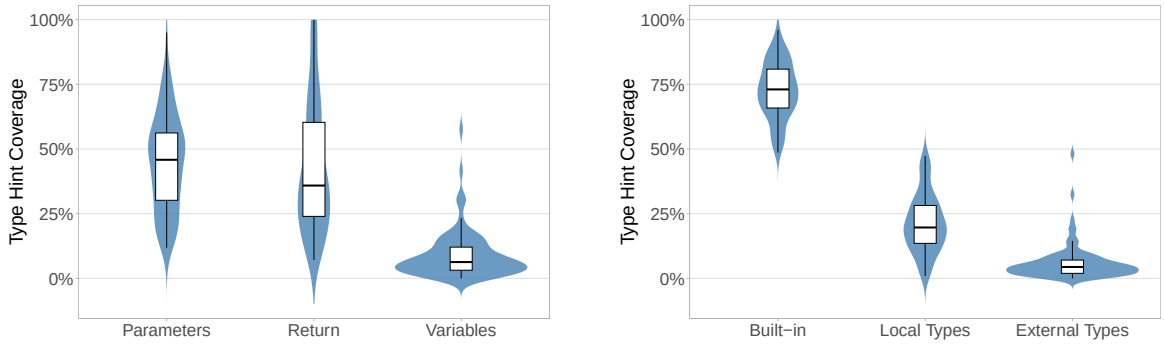
Figure 3: Distribution of overall type hint coverage.

Despite these successful cases, most projects do not apply type hints consistently across their codebase. Figure 3 shows the distribution of type hint coverage among *libraries & frameworks*. The median type hint coverage is 13.6%, while the third quartile annotated 26% of their eligible members.

> **Finding #1** Overall, 91% of *libraries & frameworks* include at least one type hint, which does not mean type hints are consistently used. On the contrary, half of the *libraries & frameworks* reported 13% of type hint coverage.

RQ2. How Type Hints are Used in Libraries and Frameworks? To address RQ2, we focused on *libraries & frameworks* with at least 10% overall type hint coverage, filtering out those with occasional usage; this resulted in a subset of 86 repositories. Figure 4 presents the distribution of the type hint coverage according to *where* they were applied, and *what* types were used in the annotations.

The distribution of type hint usage according to their location is shown in Figure 4a. As we can see, parameters and return types are significantly more annotated than variables. The median type hint coverage for parameters and return types are 45.8% and 35.9%, respectively, while variables present a median coverage of only



(a) Type Hint Location **(b) Type Hint Level**

Figure 4: Distribution of type hint usage according to their location and type levels.

6.3%. We performed a Spearman's correlation analysis to assess the relationship between the coverage of these code members [16, 21]. We found a strong correlation between the coverage of parameters and return types ($\rho = 0.64$, $p < 0.05$); by contrast, variable coverage shows a weak correlation with both parameters ($\rho = 0.27$, $p < 0.05$) and return types ($\rho = 0.28$, $p < 0.05$). This suggests that library maintainers prioritize annotating function signatures over local implementation details, probably to enhance API clarity and increase library documentation.

Figure 4b illustrates the distribution of type hint coverage according to their type levels. Built-in types are largely adopted, as 50% of the *libraries & frameworks* present 73.0% of their type hints using built-in types. On the other hand, local object types are less frequently used, with a median coverage of 19.7%. External object types are the least employed, with a median coverage of 4.4%.

Finding #2

Developers prioritize annotating function parameters and return types, indicating a focus on external API interfaces. The large majority of type hints rely on built-in types.

5 Related Work

Empirical research shows the advantage of using systems to detect errors before runtime, to reduce debugging costs, and to improve code maintainability [1, 5, 7]. Hanenberg et al. [7] have shown through a controlled experiment with 33 subjects that static typing assists developers in better navigating through the code base, reducing the effort to maintain software projects. Gao et al. [5] found that static type checkers implemented in Flow and TypeScript—supersets of the JavaScript language—could detect approximately 15% of real-world bugs in JavaScript projects. In a similar study, Bogner and Merkel [1] has shown that TypeScript applications present better code quality and legibility when compared to JavaScript ones. Mezzetti et al. [12] showed that many breaking changes detected in JavaScript projects are due to type-related issues.

Prior work has also investigated the adoption of optional and gradual type systems [3, 13, 20]. Di Grazia and Pradel [3] conducted a large-scale empirical study on type annotation evolution in Python, where they analyzed 1.4 million annotation changes extracted from 9,655 popular GitHub projects. The authors revealed that type hints are increasingly being adopted and, once added, can help developers at detecting further type errors. Similarly, Scarsbrook et al. [20] investigated TypeScript adoption among 454 repositories, finding that while TypeScript compiler is rapidly being adopted, the adoption of language-specific features varies significantly between the analyzed projects. Mir et al. [13] presents *ManyTypes4Py*, a dataset with approximately 870K type annotations, instrumented for training machine learning models specialized in performing type inference. *Unlike prior large-scale studies on Python type annotations, our work focuses on libraries and frameworks, i.e., the building blocks of modern software development.*

6 Conclusion

In this work, we report initial findings on the adoption of type hints in popular libraries and frameworks written in Python. For this, we extracted and analyzed 649,099 type annotations declared in 152 Python *libraries & frameworks* hosted on GitHub. While 9 out of 10 libraries use type hints at least once, half of them annotate, at most, 13.6% of their code members. Maintainers focus on annotating function parameters and return types, mainly using built-in types, indicating a focus on API maintenance over internal implementation details.

Next Steps. This initial study unveiled some interesting directions for research. An in-depth and qualitative analysis is needed to understand *why* developers decide to introduce type hints in certain members, as well as how these types are maintained over time. As type hints are apparently used in specific situations, new automated tools could help identify which members to annotate.

Replication Package. The dataset and scripts used in this study are publicly available at: https://doi.org/10.5281/zenodo.17615605.

Acknowledgments

This research was supported by grants from CNPq (403304/2025-3) and by FAPEMIG (APQ-02419-23).

References

[1] Justus Bogner and Manuel Merkel. 2022. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and Typescript Applications on

GitHub. In *19th International Conference on Mining Software Repositories (MSR)*. 658–669.

[2] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344.

[3] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220.

[4] Xingliang Du and Jun Ma. 2022. AexPy: Detecting API Breaking Changes in Python Packages. In *33rd International Symposium on Software Reliability Engineering (ISSRE)*. 470–481.

[5] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *39th International Conference on Software Engineering (ICSE)*. 758–769.

[6] Yimeng Guo, Zhifei Chen, Lin Chen, Wenjie Xu, Yanhui Li, Yuming Zhou, and Baowen Xu. 2024. Generating Python Type Annotations from Type Inference: How Far Are We? *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 123:1–123:38.

[7] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Software Engineering* (2014), 1335–1382.

[8] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications. In *Proceedings of the ACM on Software Engineering (FSE)*, Vol. 1. 1238–1261.

[9] Wei Jin, Dongjie Zhong, Zhen Ding, Min Fan, and Tianyin Liu. 2021. Where to Start: Studying Type Annotation Practices in Python. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 529–541. doi:10.1109/ASE51524.2021.9678947

[10] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A Systematic Review of API Evolution Literature. *Comput. Surveys* (2021), 1–36.

[11] Xuan Lin, Bin Hua, Yifan Wang, and Zhiyong Pan. 2023. Towards a Large-Scale Empirical Study of Python Static Type Annotations. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 414–425. doi:10.1109/SANER56733.2023.00046

[12] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.Js Libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP)*. 1–24.

[13] Amir M. Mir, Evaldas Latoskinas, and Georgios Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-based Type Inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 585–589.

[14] João Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying Experts in Software Libraries and Frameworks Among GitHub Users. In *16th International Conference on Mining Software Repositories (MSR)*. 276–287.

[15] João Eduardo Montandon, Luciana Lourdes Silva, Cristiano Politowski, Daniel Prates, Arthur de Brito Bonifácio, and Ghizlane El Boussaidi. 2025. Unboxing Default Argument Breaking Changes in 1 + 2 Data Science Libraries. *Journal of Systems and Software* (2025), 1–38.

[16] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and Use of Java Generics. *Empirical Software Engineering* 18, 6 (2013), 1047–1089.

[17] Luciano Ramalho. 2022. *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media.

[18] Amanda Santana, Eduardo Figueiredo, and Juliana Alves Pereira. 2024. Unraveling the Impact of Code Smell Agglomerations on Code Stability. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 461–473.

[19] Rizel Scarlett. 2023. Why Python Keeps Growing, Explained.

[20] Joshua D. Scarsbrook, Mark Utting, and Ryan K. L. Ko. 2023. TypeScript's Evolution: An Analysis of Feature Adoption Over Time. In *20th International Conference on Mining Software Repositories (MSR)*. 109–114.

[21] Peter Sprent and Nigel C. Smeeton. 2016. *Applied Nonparametric Statistical Methods*. CRC Press.

[22] Stack Overflow. 2024. Stack Overflow Developer Survey.

[23] Valerio Terragni, Partha Roop, and Kelly Blincoe. 2024. The Future of Software Engineering in an AI-Driven World. In *International Workshop on Software Engineering in 2030*. 1–6.

[24] Evgenia Verbina. 2025. The State of Python 2025 | The PyCharm Blog. https://blog.jetbrains.com/pycharm/2025/08/the-state-of-python-2025/.

[25] Jiawei Wang, Tzu-Yang KUO, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 138–149.

[26] Matthijs J Warrens. 2015. Five Ways to Look at Cohen's Kappa. *Journal of Psychology & Psychotherapy* (2015), 1–4.

[27] Adam Brooks Webber. 2002. *Modern Programming Languages: A Practical Introduction*. Franklin Beedle & Assoc.

[28] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. 2021. Unveiling the Mystery of API Evolution in Deep Learning Frameworks: A Case Study of Tensorflow 2. In *43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 238–247.