The Evolution of Type Annotations in Python: An Empirical Study

Luca Di Grazia
University of Stuttgart
Germany
luca.di-grazia@iste.uni-stuttgart.de

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

ABSTRACT

Type annotations and gradual type checkers attempt to reveal errors and facilitate maintenance in dynamically typed programming languages. Despite the availability of these features and tools, it is currently unclear how quickly developers are adopting them, what strategies they follow when doing so, and whether adding type annotations reveals more type errors. This paper presents the first large-scale empirical study of the evolution of type annotations and type errors in Python. The study is based on an analysis of 1,414,936 type annotation changes, which we extract from 1,123,393 commits among 9,655 projects. Our results show that (i) type annotations are getting more popular, and once added, often remain unchanged in the projects for a long time, (ii) projects follow three evolution patterns for type annotation usage – *regular annotation*, *type sprints*, and *occasional uses* – and that the used pattern correlates with the number of contributors, (iii) more type annotations help find more type errors (0.704 correlation), but nevertheless, many commits (78.3%) are committed despite having such errors. Our findings show that better developer training and automated techniques for adding type annotations are needed, as most code still remains unannotated, and they call for a better integration of gradual type checking into the development process.

CCS CONCEPTS

• **Software and its engineering** → **Language features**; **Software evolution**; • **General and reference** → Empirical studies.

ACM Reference Format:
Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *Proceedings of The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Dynamically typed languages, such as Python and JavaScript, have become the most popular languages for newly written code.[1] One reason for this popularity is their lightweight syntax, which does not require developers to specify the types of parameters, return values, fields, or variables. At the same type, the absence of static type annotations often hampers maintenance, causes type-related bugs to be missed, and limits IDE support.

The problems caused by the complete absence of type annotations has motivated optional type annotations. They offer a flexible middle ground between no type annotations at all and a fully statically typed language, enabling each developer to annotate only those types she believes to be beneficial. The two most popular dynamically typed languages, Python and JavaScript, both support optional type annotations. In particular, Python 3.5 specifies the meaning of type annotations for functions,[2] and Python 3.6 adds syntax for specifying the types of variables.[3]

(a) Commit 1.

```
def f(x: float, y):
  sum = x + y
  if (sum % 2) == 0:
    return True
```

(b) Commit 2.

```
def f(x: int, y) -> bool:
  sum: int = x + y
  if (sum % 2) == 0:
    return True
```

(c) Commit 3.

```
def f(x: int, y) -> Optional[bool]:
  sum: int = x + y
  if (sum % 2) == 0:
    return True
```

Figure 1: Example of an evolving, partially type-annotated Python function.

In recent years, a variety of tools have been proposed for helping developers deal with type annotations in dynamically typed languages. Gradual type checkers [35, 46] use the available type annotations, possibly along with type information for popular libraries, to check for type errors. Beyond gradual type checkers, recent work proposes techniques to infer and predict type annotations, based on static analysis [3, 11, 22, 25], dynamic analysis [2, 41], probabilistic rules [52], learned predictive models [1, 23, 30, 40], and combinations of the former [38]. Such tools help annotate code with types, in particular code written before the standardized introduction of type annotations into the programming language.

Figure 1 shows an example of a partially type-annotated Python function and its evolution across three commits, which illustrates different kinds of type annotations, how they may evolve, and type errors that may become apparent as a result. The function expects two parameters and returns whether their sum is an even number. In Commit 1, the first parameter is initially annotated to be of type `float`, whereas the second parameter remains unannotated. Commit 2 updates the parameter type of `x` from `float` to `int`, and also inserts an annotation of the return type being `bool`. Moreover, the code change inserts an annotation for the `sum` variable to be of type `int`. Type checking Commit 2 will produce a type error because not all paths through the function return a boolean, but the function implicitly returns `None` when the sum is not even. Finally, Commit 3 fixes the type error by modifying the return type annotation to `Optional[bool]`.

Several years have passed since the release of Python 3.5 in 2015 and type annotations are hypothesized to be useful to developers, but it is currently unclear how often they are adopted in practice and how this trend is evolving. How frequently do people use this feature? And how has its usage changed over time? This serves to

[1] https://octoverse.github.com/

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

[3] https://www.python.org/dev/peps/pep-0526/

double check if adding type information to the Python language is actually perceived as a benefit worth undertaking by developers. Moreover, the gradual type system of Python defines several standard library types such as `int`, `List`, and `Optional`. Which of those are the most useful to developers? And which of those have the most changes? We can find that adding those types is non-trivial, and they sometimes result in incorrect type annotations. What impact do the annotations have on detected type errors, and if errors are detected, do developers address them? This helps to understand if type checkers are useful during the development process. As a result, answering these and other questions consist in better understanding the adoption of type annotations in Python, identify issues that developer's commonly face in this process, and steer future research on developer tools toward the most relevant problems.

This paper presents the first comprehensive study of the evolution of type annotations and type errors in Python. The study is performed on 9,655 of the most popular Python projects, analyzes 1,123,393 commits, and studies 668 projects in more detail, as they contain at least one annotation. We address four research questions:

- *RQ1: How does the adoption of type annotations evolve at the ecosystem level?* To better understand to what extent type annotations are getting adopted by the developer community as a whole, we study the evolution of the prevalence of type annotations across a wide range of projects.
- *RQ2: How does the usage of type annotations evolve at the project level?* This question aims at understanding the evolution of type annotations based on a commit-based timeline of single projects.
- *RQ3: How do individual type annotations evolve?* Answering this question helps understand if and how type annotations, once inserted, change over time. We also study whether type annotations are added alongside other code changes or in specific commits, and how long they remain in a code base.
- *RQ4: How do statically detectable type errors evolve and how do they relate to the type annotations in a project?* This question aims at understanding to what extent gradual type checkers can help avoid type errors, whether developers fix these errors, and how they are impacted by adding, changing, or removing annotations.

Our methodology is based on an AST-based analysis to extract type annotations, a differential analysis to understand how type annotations evolve over time, and gradual type checking on different versions of the projects.

Prior work has studied other aspects of dynamically typed programming languages, e.g., whether developers migrate from Python 2 to Python 3 [50], how linters are used in JavaScript [45], and whether type annotations in JavaScript [12] and Python [28] reveal bugs. Ore et al. assess the human effort involved in adding type annotations[34], and Hanenberg et al. study the impact of type annotations on development time [17] and maintainability [19]. Another recent study [39] is about the kinds of type annotations developers use and how the errors reported by different gradual type checkers differ. In contrast to all the above, our study focuses on the evolution of types and type errors over time, allowing us to better understand the long-term trends in adopting gradual typing.

Our study leads to several findings regarding the prevalence, characteristics, and evolution of type annotations and type errors in real-world Python code, including:

- Type annotations are getting more and more popular, but are still far from being the norm. Less than 10% of all possible code elements are currently annotated, even in projects that have at least one type annotation. This trend is slowly changing in favor of more annotations, offering a huge opportunity for researchers and practitioners to build tools that help with this process.
- Most type annotations are added alongside other code, but developers also occasionally (1.3% of all type-editing commits) work on the type annotations only.
- Developers mostly focus on annotating parameter types and return types, and less on variable types.
- Once added, many type annotations are never updated (90.1%) and many type annotations (70.4%) are still present in the latest version of a project, rewarding the effort of adding type annotations. If developers change type annotations, then optional types are commonly involved in the change.
- Most commits (78.3%) contain statically detectable type errors but are nevertheless integrated into the code base.
- Adding type annotations tends to increase the number of detected type errors (correlation of 0.704).

Our findings have several implications for developers and researchers working on developer tooling. First, we find that adding type annotations is a long-term investment because they are rarely modified, which can impact the maintainability of a code base over years. Second, the result that more than 90% of program elements are not yet annotated, both in legacy code and newly written code, is a call to arms for creating tools that infer and predict types. Recent work on learning-based type prediction is a promising step [23, 30, 38]. Third, the repetitive nature of type annotation changes pinpoints several easy to avoid mistakes, such as avoiding corner cases using a type `T` instead of its optional variant `Optional[T]`, as in our motivating example. Finally, the fact that most commits have statically detectable type errors calls for more developer awareness and better integration of gradual type checkers into the development process.

In summary, this paper makes the following contributions:

- A comprehensive study of the evolution of type annotations and type errors in real-world Python code.
- Findings that may impact developers and teachers, as well as future work on tools and techniques for developers.
- A dataset of type annotation-related code changes to be used in future work, e.g., on mining and learning from these changes.

Our implementation, the dataset, and all experimental results are available online.[4]

2 METHODOLOGY

This section discusses the methodology we use to address our research questions.

2.1 Extracting and Studying Type Annotations

The core concept of our study are type annotations:

Definition 1 (Type annotation)**.** A type annotation is a tuple $t_{ann} = (t, n, k, l)$, where t is a type, n is the name of an annotated

[4] https://github.com/sola-st/PythonTypeAnnotationStudy

program element, k is the kind of the annotated program element, and l is the code location of the type annotation.

In line with the type annotations specified by the Python language, we consider three kinds k of program elements: argument types, return types, and variable types. The code location l is specified by a file path and a line number. As an example, consider the code in Figure 1a. The type annotation for the annotated parameter type is (t, n, k, l), where t is `float`, n is `x`, k is "argument type", and l is line 1 of the corresponding file.

Given the type annotations in a code base, we compute the following notion of coverage, which indicates how many of all program elements that could be annotated are indeed annotated:

Definition 2 (Type annotation coverage)**.** Given a code base B, the type annotation coverage for a kind k_{target} of program elements is:

$$cov_{ann} = \frac{|\{(t, n, k, l) \in B \mid k = k_{target}\}|}{|\{\text{All program elements of kind } k_{target}\}|}$$

For example, consider a code base that consists only of the function in Figure 1b. The type annotation coverage for argument types is 50%, because one out of two arguments is annotated, whereas the type annotation coverage for return types is 100%.

To extract both present and missing type annotations, we perform an AST-based static analysis of each Python file (*.py*) and each Python stub file (*.pyi*) in a version of a project. The analysis visits each node that corresponds to a possibly type-annotated program element and, if an annotation is found, extracts the corresponding tuple. We focus on type annotations in the type definition syntax added in Python 3.5 (PEP 484) and Python 3.6 (PEP 526). In contrast, we do not consider types described in informal type comments, because type comments are only partially supported by tools and because our preliminary results found type comments to occur clearly less often than annotations in proper syntax. The analysis of Python files and Python stub files is implemented on top of the LibCST and Typed AST libraries, respectively.[5]

2.2 Extracting and Studying Type Annotation Changes

As the primary focus of our work is to study the evolution of type annotations, we extract annotations across the history of a project and relate them to each other.

Definition 3 (Type annotation change)**.** A type annotation change is a tuple $t_{change} = (t^{old}_{ann}, t^{new}_{ann}, c, d)$, where t^{old}_{ann} and t^{new}_{ann} are the type annotations before and after a code change, respectively, c is the kind of code change, and d is the date of the code change.

We consider three kinds of code changes: inserting, updating, and removing a type annotation. In a type annotation change that inserts or removes an annotation, the old or new annotation is undefined, respectively, which we represent with `_`. For example, the commits in Figure 1 involve four type annotation changes: an update of the argument type `x`, a newly inserted annotation for the return type of `f`, a newly inserted annotation for the variable `sum`, and an update of the return type of `f`.

To study the evolution of individual type annotations, we combine multiple type annotation changes into a history:

[5]https://github.com/Instagram/LibCST, https://github.com/python/typed_ast

Algorithm 1 Extract type annotation changes from commits.

Input: Sequence C of commits
Output: Set T of type annotation changes (old and new type pairs)

```
1:  T ← ∅
2:  for commit c in C do
3:      T_old ← type annotations in code before commit c
4:      T_new ← type annotations in code after commit c
5:      d ← date of c
6:      T' ← ∅
7:      for (t_old, t_new) ∈ T_old × T_new do
8:          if ∃ hunk h in c where t_old in oldLineRange(h)
9:          and t_new in newLineRange(h) then
10:             if t_old and t_new have same kind k
11:             and same name n then
12:                 Add (t_old, t_new, "update", d) to T'
13:     T' ← ensureSingleMatch(T')
14:     for t_new not yet added to T' do
15:         Add (_, t_new, "insert", d) to T'
16:     for t_old not yet added to T' do
17:         Add (t_old, _, "remove", d) to T'
18:     T ← T ∪ T'
```

Definition 4 (Type annotation history)**.** A type annotation history is a sequence $[t^1_{change}, ..., t^m_{change}]$ of type annotation changes, where:

- the t^1_{change} has code change kind c ="insert",
- there is at most one type annotation change with c ="remove" and if it exists, then it is t^m_{change},
- the kind k of program element is the same in all type annotation changes, and
- for consecutive type annotation changes, the new type annotation t^{new}_{ann} of the first change is the same as the old type annotation t^{old}_{ann} of the second change.

For example, consider an extended version of the example in Figure 1 where the code change in the figure is preceded by a commit that adds the function without any type annotations and followed by a commit that removes the annotation of the `x` argument again. This evolution of the argument type would be represented as a type annotation history with three type annotation changes, which describe how the annotation of argument `x` gets inserted, updated, and removed, respectively.

We compute type annotation changes and type annotation histories by combining the annotations extracted by our AST-based analysis across the commit history of a repository. Tracking annotations across histories is a non-trivial challenge, e.g., because line numbers change due to removed and added code, or because developers may modify multiple type annotations in a single commit.

Algorithm 1 summarizes our approach for addressing this challenge. The algorithm iterates through a sequence of commits and extracts a set of type annotation changes from it. At first, lines 3 and 4 extract the type annotations from the old and the new version of a commit, respectively. Then, the algorithm compares these annotations based on their code location, the kind of the annotated element, and the name of the annotated element. The goal is to find

matches, i.e., pairs of an existing annotation and a revised version of it, and that hence, should be combined into a type annotation update. To this end, the algorithm builds upon the concept of hunks, i.e., consecutive lines that are changed together. Concretely, lines 8 to 13 check whether a pair of an old and a new type annotation fit into the line range of a hunk in the commit, and if so, compares the kind and name of the annotated program element. Because we collect the life of type annotations from insertion to removing (if removed), the *ensureSingleMatch* function checks if it is an update of a program element already collected or if it is a different program element without creating duplicate elements. After finding pairs of type annotations that are changed in the commit, lines 14 to 17 consider annotations that exist only in the old or only in the new version of the commit. These annotations are added to the set of type annotation changes as "inserted" and "removed" annotations, respectively.

Because Algorithm 1 is heuristic, we validate the accuracy of the type annotation changes it extracts by manually inspecting 85 histories with a total of 204 type annotation code changes. We randomly sample these type annotation histories based on three categories: (i) annotations that are never updated and still present in the last analyzed version of the project, (ii) annotations that are never updated but removed at some point during the commit history, (iii) annotations that are updated multiple times. For each sampled history, we carefully inspect the commits involving the annotation and establish a ground truth history. We find that 90.1% of the automatically extracted histories match the manually established ground truth, i.e., the vast majority of histories is correctly extracted. The main reasons for (partially) incorrect type annotation histories are mismatched annotations due to multiple identifiers with the same name in the same file, and renamed or deleted files that our analysis does not track.

2.3 Gathering and Studying of Type Errors

We study the evolution of type errors by running a gradual type checker on different commits in the history of a project. There are several popular type checkers for Python, e.g., pyre, mypy, and pytype. For our study, we focus on pyre, because it is industrially used, e.g., at Facebook, and could successfully analyze the studied projects.

The kinds of type errors reported by pyre and other gradual type checkers fall into two categories. One category of errors are those caused by missing dependencies, e.g., when the type checker cannot find an imported class or cannot resolve a reference to a type. These errors are unlikely to occur when a type checker is used by the project developers, assuming that the developers create a proper configuration that resolves all external dependencies. In contrast, eliminating these errors in a large-scale study is difficult because resolving all dependencies and configuring the type checker to find the dependencies is non-trivial. The second category of errors are the actual type errors, which result from inconsistencies between inferred and annotated types of values and the uses of these values. For example, these errors occur because a function argument is incompatible with the declared parameter type or because a method overrides another method with an incompatible type signature. We focus our study on the second category of errors, and unless otherwise mentioned, ignore the first category, providing a realistic view of what errors the developers of a project would see when using a type checker.

2.4 Selection of Projects to Study

As subjects for our study we select a wide range of open-source projects based on their creation time and their popularity. At first, we gather the list of all Python projects at GitHub via the GitHub API.[6] We group the projects by their creation date, considering projects created in the years 2010 to 2019, into ten groups that each cover one year. Then, we sort the projects in each group by their number of stars and select the top-1000 per group, which yields a total of 10,000 projects to study. The rationale for first grouping and then sampling is to avoid biasing our study toward projects created in a particular time frame, e.g., mostly old projects. Removing projects that we could not clone, e.g., because they became unavailable since the beginning of our study, the total number of analyzed repositories is 9,655.

3 RESULTS

This section presents the results we obtain when addressing our four research questions. Before going through the research questions, we give an overview of the analyzed data. In total, the study involves 1,123,393 commits in 9,655 repositories. Our analysis extracts 1,414,936 type annotation changes from these commits. These type annotation changes correspond to 61,861 commits and 668 repositories that have at least one type annotation change. Our results are for these 668 projects. As general statistics, the number of commits with at least one type annotation grows every year. An early adoption started already in 2015, where 3.8% of the commits contain at least one type annotation and this number of commits grows every year until reaching 10.9% in 2021.

3.1 RQ1: Ecosystem-level Evolution of Type Annotations

To understand whether type annotations are becoming more common in the Python ecosystem as a whole, we analyze the evolution across all studied projects. The goal is to understand trends in the ecosystem, e.g., caused by the introduction of new programming language features or tools. We perform this analysis from two points of view. First, we analyze the evolution of the absolute number of type annotations. Second, we measure the evolution of type annotation coverage.

3.1.1 How is the total number of type annotations evolving? We measure the overall number of type annotations and the overall number of lines of code between 2015 and 2021. Figure 2 shows the results, taking a snapshot of each project on October 1 of each year. The main observation is that both the absolute number of type annotations and the number of type annotations per line of code are steadily increasing in a roughly linear manner since 2017. In 2021, there are around 50.1 annotations per 1,000 lines of code.

To better understand the relative importance of annotations provided in regular Python files (*.py*) and Python stub files (*.pyi*), Figure 2 distinguishes between them. It shows that the vast majority,

[6] https://api.github.com/search/

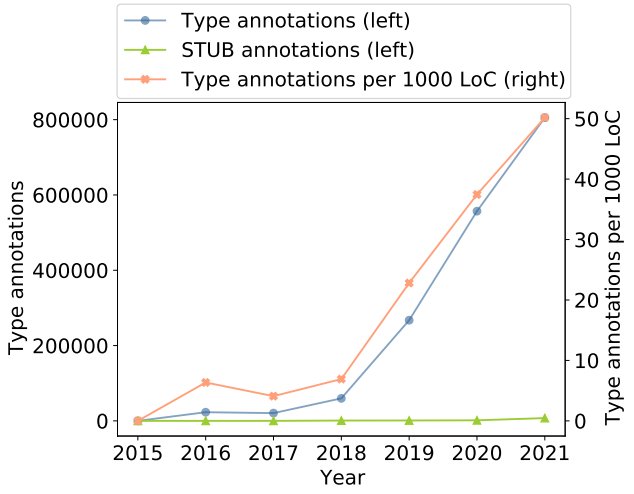

Figure 2: Evolution of type annotations across all projects.

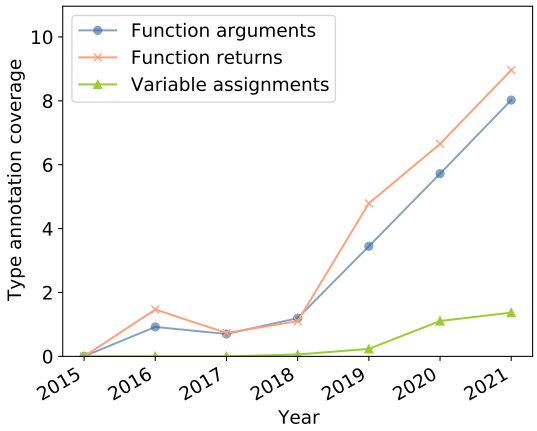

Figure 3: Evolution of program elements with and without type annotations.

e.g., 99.1% of all annotations present in 2021, are provided in regular Python files. A manual inspection of 20 stub files sampled from 13 projects shows that most annotations provided in stub files are about APIs of external libraries (17 out of the 20 files), for example, native libraries accessed via Python's native bindings. Two of the remaining three files are automatically generated. Given the relatively low number of annotations in stub files and their focus on external libraries, the remainder of the study considers only annotations in regular Python files.

Finding 1: Type annotations are getting more and more popular, with an increase of about 15 type annotation per 1,000 line of code after 2017 and reaching 50.1 annotations per 1,000 lines of code in 2021.

3.1.2 How is the type annotation coverage evolving? This question is important to understand how much of the available "annotation potential" developers are currently using. Out of all 9,655 projects we study, only 668 (7%) use type annotations at all. That is, six years after the introduction of type annotations into the language, the large majority of projects is not yet using this feature. Figure 3 takes a detailed look into those 668 projects that use type annotations. The figure shows the type annotation coverage for function arguments, return values, and variables on October 1 of each year. The results allow for several observations. First, the type annotation coverage is steadily increasing. Second, developers prefer to annotate function argument types and return types, but focus less on variable types. Third, despite the clear upward trend, the type annotation coverage is still relatively low, with an average of around 8% for function arguments and return values.

To put the type annotation coverage in perspective, we consider a project known for its heavy use of type annotations: *mypy*[7], which is one of the gradual type checkers for Python. This project has a type annotation coverage of 62.2% for parameter types, 94.9% for return types, and 23.4% for variable types. A manual inspection shows two main reasons for leaving program elements unannotated. First, the developers do not annotate `self` parameters, as `self` always has the type of the current class, and hence, does not really need a type annotation. Second, a significant number of unannotated local variables have types that can be easily inferred by a gradual type checker, e.g., because a variable is assigned the result of a constructor call or the variable is assigned an annotated variable. Omitting such annotations fits the philosophy behind gradual typing, i.e., to annotate types when it is helpful without cluttering the code with unnecessary annotations.

Finding 2: Type annotations are not yet the norm, with less than 10% of all possible code elements being currently annotated, but there is a clear upward trend. Function arguments and return types are annotated more commonly than variables.

3.2 RQ2: Project-level Evolution of Type Annotations

After considering the Python ecosystem as a whole in RQ1, we now study the evolution of type annotations within individual projects. To this end, we measure how many type annotations a project has at different points during its lifetime, where lifetime means all commits from creating the project until the end of our measurement period (end of 2021). Putting the absolute number of type annotations in perspective, we also measure the number of lines of code at each point during the project lifetime. A commonality of almost all studied projects is that the number of type annotations is rarely decreasing, but instead grows continuously, i.e., once annotations are added, developers rarely remove them.

By inspecting the evolution of type annotations of various projects, we identify three common evolution patterns, illustrated in Figure 4 with three representative projects. For each project, the plot shows how the code size and the number of type annotations have evolved throughout the project's history.

[7] https://github.com/python/mypy

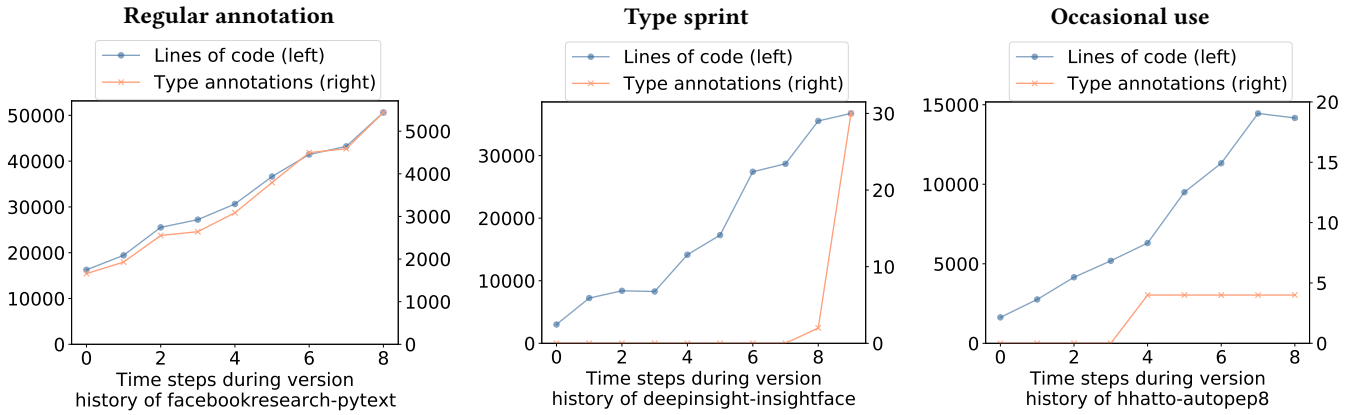

Figure 4: Per-project evolution of three representative projects.

- *Regular annotation.* Some projects, such as facebookresearch-pytext,[8] have adopted type annotations throughout their entire history and regularly add annotations as the project is growing. The typical evolution pattern of these projects is that the number of type annotations is growing at roughly the same rate as the overall code size. As can be observed by comparing the absolute numbers of lines of code (left axis in Figure 4) and type annotations (right axis), such projects often have significantly more type annotations than the average project. For the specific example, there are about 100 annotations per 1,000 lines of code, whereas the average project reaches, even in 2021, only about 30 annotations per 1,000 lines of code (Figure 2).
- *Type sprints.* Some other projects, e.g., deepinsight-insightface,[9] have invested into type annotations during a focused, sprint-like effort, where many annotations are added at once, but otherwise do not regularly add annotations. A variant of this pattern is a step-like curve of the number of type annotations, i.e., projects that add annotations in multiple yet non-continuous efforts.
- *Occasional use.* Some projects, such as hhatto-autopep8,[10], have only a small number of annotations, typically added in a single or very few files. This kind of project is included into the study because we consider all projects with at least one type annotation.

To measure the prevalence of these three patterns across all studied projects, Algorithm 2 heuristically determines whether a project fits any of the patterns. The algorithm divides the commit history of a project into ten equally sized steps, and it then checks the number of annotations present at each step and the slope from the previous to the current time step. For example, if the average slope across all time steps is positive, then the algorithm classifies the project as "Regular annotation", whereas a project with four or more time steps that do not add any annotation is classified as "Occasional use". In the algorithm, $P.slope$ refers to the list of slopes observed at different time steps, and $P.slope(0.0)$ checks how many of these slope values are equal to zero. We validate Algorithm 2 in three steps. First, we generate the evolution plots of all studied projects. Second, we manually inspected 35 randomly selected plots and, looking at the curve, manually label each of them. Third, we run Algorithm 2 and compare the labels produced by the algorithm with our manual labels. During this validation, the algorithmically produced labels all match our manual labeling.

Algorithm 2 Determine project-level evolution pattern.

Input: Project P
Output: Evolution pattern of P

Divide P into 10 time steps of equal number of commits
$P.annotations \leftarrow$ Number of annotations for each time step
$P.slope \leftarrow$ Annotation evolution slope for each time step
if $max(P.annotations) < 15$ **or** $P.slope.count(0.0) > 8$ **then**
    return "Occasional use"
else if $P.slope.count(0.0) >= 4$ **then**
    return "Type sprints"
else if $Average(P.slope) >= 0$ **then**
    return "Regular annotators"
else
    return "Other"

Running the algorithm across all 668 projects shows that 44.4% perform "Regular annotation", 28.1% use "Type sprints", and 25.4% are "Occasional use". The remaining 2.1% are "Other", i.e., their evolution does not fit any of the three patterns.

We also study the relation between the patterns and the characteristics of the project, such as the number of stars and contributors. While most characteristics are independent of a project's evolution pattern, we find a relationship with the number of contributors. "Regular annotation" projects have an average of 62 contributors, projects using "Type sprints" have 45 contributors, and projects with "occasional use" have only 25 contributors, on average. These numbers show that regularly adding type annotations is practice followed particularly in large repositories with a more solid organization, presumably because type annotations help coordinate between a large number of developers.

[8] https://github.com/facebookresearch/pytext
[9] https://github.com/deepinsight/insightface
[10] https://github.com/hhatto/autopep8

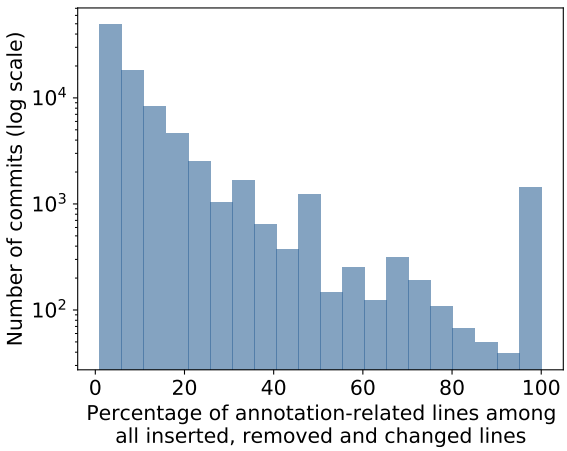

Figure 5: Percentage of annotation-related, edited lines among all edited lines.

Finding 3: Most projects follow one of three evolution patterns when adding type annotations: "regular annotation", "type sprints", and "occasional use". Projects with more contributors tend to use "regular annotation", whereas projects with few contributors tend to follow "occasional use".

3.3 RQ3: Evolution of Individual Type Annotations

The following studies how individual type annotations evolve, which allows us to better understand how developers insert, modify, and remove type annotations.

3.3.1 When do developers edit types? Since type annotations are optional in Python, developers can freely choose when to add or edit them. In particular, a developer can add new type annotations along with other code, e.g., along with a newly added function, or in a separate step later on, e.g., as part of a code improvement session. To understand when developers insert or modify types in a code base, we analyze all commits in the dataset that affect at least one type annotation. For each such commit, we compute how many of all lines edit a type annotation. The resulting value is a percentage, where 100% means that the commit is only to edit type annotations, and a value closer to 0% means that more other code is edited alongside the type annotation edit.

The results are shown in Figure 5. We see a bi-modal distribution, where the majority of commits edit a significant number of other lines in addition to editing type annotations. At the same time, there are a non-negligible number of commits that exclusively edit type annotations, showing that developers at least sometimes specifically focus on editing type annotations.

Finding 4: Most type annotations are edited alongside other code, but developers also occasionally (1.3% of all type-editing commits) work on the type annotations only.

(a) Commit 1.

```
def pdist2(X: torch.Tensor,
    Z: torch.Tensor = None,
    order: PDist2Order = PDist2Order.d_second)
    -> torch.Tensor:
```

(b) Commit 2.

```
def pdist2( X,
        Z = None,
        order = PDist2Order.d_second):
# type: (torch.Tensor, torch.Tensor,
# PDist2Order) -> torch.Tensor
```

Figure 6: Example of removing a type annotation.

3.3.2 How long do type annotations remain in a code base? Answering this question helps understand whether adding type annotations is a long-term investment. We address the question in two ways. At first, we study how many of all ever added type annotations are still present in the latest version of the projects. To this end, we compute for each repository the number of type annotation changes that insert an annotation. In addition, we analyze the latest version of each repository, cloned on March 7, 2022, and compute how many type annotations it contains. In absolute values, 70.4% of all annotations "survive" until the latest version of a repository. For some projects, shown in the upper-right corner of the figure, all ever added annotations are still present in the latest version.

Second, we consider all type annotation histories in the dataset where the last change is a commit that removes the annotation. We compute the lifetime of each such annotation as the difference between the first and the last date in the history. In total, we find that 29.6% of all type annotations eventually get removed. We analyze in detail the removed type annotations. Their average lifetime is 160 days, showing that even annotations that get removed remain in the code for a while. An example is shown in Figure 6, where the types are removed from the source code and saved in a comment.[11] The commit messages says that the developers are adding support for Python 2.7, which does not support the type annotation syntax yet. We inspect a random sample of 25 of all removed annotations and classify them into three categories. We find that in 48% of the cases the entire files are removed or renamed, in 40% of the cases the program element is removed or renamed, and in 12% of cases types are explicitly removed, e.g., for supporting Python 2 or to simplify the code as shown in Figure 6. As a result, we can affirm that type annotations are a long-term investment because only in very few cases types are explicitly removed again.

Finding 5: 70.4% of all ever inserted type annotations are still present in the latest version of a repository, and those type annotations that get removed at some point "live" for an average of 126 days.

3.3.3 (How) do type annotations change? Once type annotations are added, developers may modify them, e.g., to fix a wrong annotation or because the annotated code is evolving. In this research question we do not consider types that are removed. We study type annotation changes by, at first, investigating how often type annotations are updated at all. To this end, we analyze all extracted type annotation histories and compute how many updates of a type annotation they contain.

[11] https://github.com/erikwijmans/Pointnet2_PyTorch/commit/a89c4d1

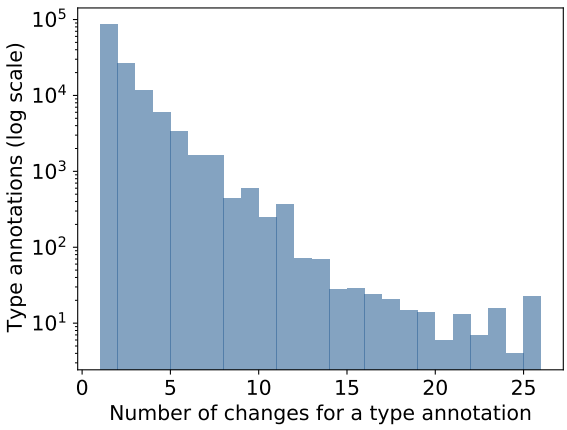

Figure 7: Number of times that the same type annotation is updated by developers. Not shown are the 90.1% of all type annotations with zero updates.

Figure 7 shows how many type annotations we find that are updated a specific number of times. The plot does *not* show the vast majority (90.1%) of all type annotations that are never updated. Overall, we count 139,586 annotation updates, with an average of 1.8 updates for each type annotation that gets updated at least once. The maximum number of observed updates is 25, which is an outlier though. Out of those annotations that get updated at all, most are updated five times or less. Figure 8 shows an example. In this case the type annotation is updated to `Sequence`,[12], and later to the user type `ModelField`.[13]

To better understand how annotations that get updated evolve, we analyze which kind of type annotation updates are most common. Figure 9 shows the results of this analysis, where the three plots show the five most commonly observed updates for argument types, return types, and variable types, respectively. We show all types that are part of the Python language and its standard library as-is, e.g., `str` and `Optional[int]`, and abstract all user-defined types into `UserType`. The results allow for two observations. First, many type annotation updates involve custom types. Second, many updates affect optional types, e.g., changing `str` to `Optional[str]`. In total, we count 14,750 type annotation updates involving optional types. A manual inspection of some of these updates shows that developers easily get confused about whether a parameter is optional or whether a variable should be immediately initialized. Figure 10 shows an example, where the code on the left is type-incorrect, which the developer then fixed.[14]

Finding 6: Most type annotations (90.1%) never get updated. For those that get updated, a frequent update pattern involves custom and optional types, which seems a common source of confusion.

[12]https://github.com/tiangolo/fastapi/commit/c20c9d8
[13]https://github.com/tiangolo/fastapi/commit/f7b7ed0
[14]https://github.com/gogcom/galaxy-integrations-python-api/commit/18f6cd7

3.4 RQ4: Type Errors vs. Type Annotations

In this last research question, we inspect the number of type errors and their relationship with type annotations. We divide this analysis into three parts. First, we compute how many type errors are in these repositories. Second, we check if there is a correlation between the number of type errors and type annotations. Third, we analyze if insertions of type annotations increase the number of type errors. These three parts are performed on all the 668 projects that have at least one annotation.

3.4.1 How common are type errors? This research question is important to understand what value gradual type checkers could add to real-world Python projects and whether today's developers are using these tools. For each analyzed project, we run the type checker on each commit in the project's history and then count the number of non-dependency-related type errors (Section 2.3). We find that 78.3% of the analyzed snapshots have at least one type error. On average, there are 6 type errors per 1000 lines of code. This result indicates that type checking is not yet part of the typical development routine, calling for better tool support and more developer awareness.

To better understand the kinds of detected type errors, we analyze what kinds of errors are most common in the most recent versions of the studied projects. We find a total of 90,871 type errors and that a few kinds of errors occur repeatedly, in particular the error *incompatible variable types* (17.6%) and *incompatible parameter types* (12.6%).[15]

Finding 7: Most projects have statically detectable type errors, and type errors seem to not prevent developers from committing code. A few kinds of mistakes account for most type errors.

3.4.2 How does the number of type errors depend on the number of type annotations? One major goal of introducing type annotations is to statically detect otherwise missed type errors. The reason is that most gradual type checkers, including the pyre tool used here, run additional type checks if more annotations are present. Even if these tools are not perfect, several studies proved the usefulness of type checkers [12, 28], so we decide to use pyre for this research question. To check if type annotations indeed provide this benefit in practice, we compute the correlation between the number of type errors and the number of type annotations in each project. Figure 11 visualizes the relation between these two measures, showing that there is a significant correlation (Pearson coefficient of 0.704) between them. We conclude that adding type annotations is only the first step toward improving type correctness, and the developers also need to introduce type checking into their developing routine.

Finding 8: Adding type annotations positively correlates with an increase in the number of detected type errors (correlation: 0.704). Developers should introduce type checks in their developing routine to find and fix such errors early on.

[15]https://pyre-check.org/docs/errors/#error-codes

(a) Commit 1.

```
def request_params_to_args(
        required_params: List[Field], ...
) -> Tuple[Dict[str, Any], List[ErrorWrapper]]:
```

(b) Commit 2.

```
def request_params_to_args(
        required_params: Sequence[Field], ...
) -> Tuple[Dict[str, Any], List[ErrorWrapper]]:
```

(c) Commit 3.

```
def request_params_to_args(
        required_params: Sequence[ModelField], ...
) -> Tuple[Dict[str, Any], List[ErrorWrapper]]:
```

Figure 8: Example of a type annotation updated multiple times.

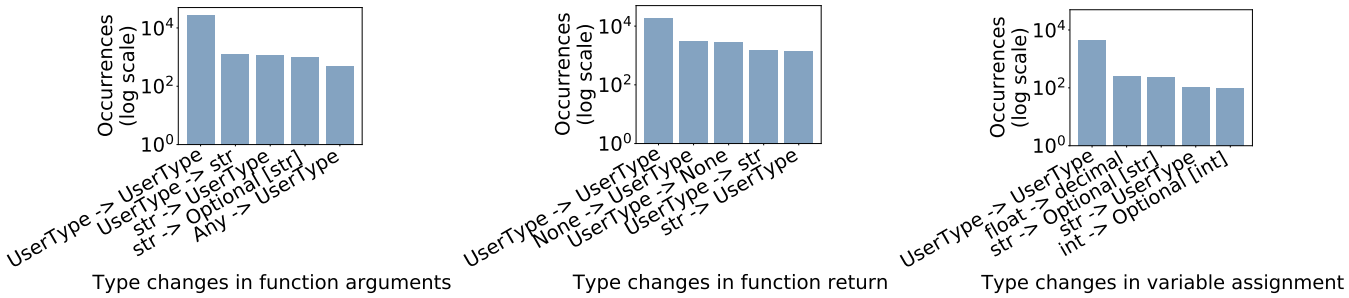

Figure 9: Most common kinds of type annotation changes.

(a) Commit 1.

```
class LicenseInfo():
        license_type: str
        owner: str = None
```

(b) Commit 2.

```
class LicenseInfo():
        license_type: str
        owner: Optional[str] = None
```

Figure 10: Example of adding a wrong type annotation and then updating it with *Optional* type.

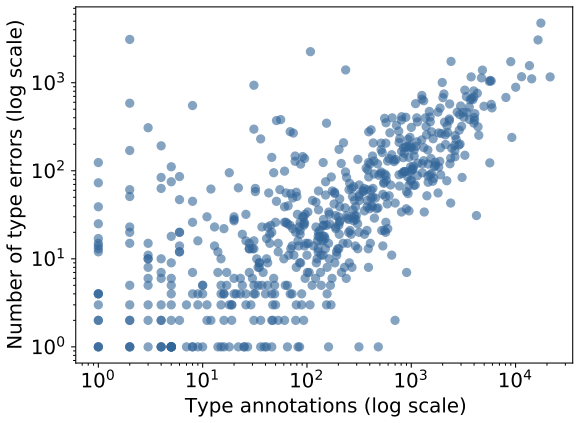

Figure 11: Relation between type errors and type annotations in a project (correlation: 0.704).

3.4.3 How does the number of type errors evolve when type annotations evolve? The following aims to understand how evolving the type annotations of a project impacts the statically detectable type errors. For this purpose, we extract from all type annotation changes only those where all lines changed in the commit correspond to only adding type annotations, which we call *pure commits*. Pure commits are interesting because they allow us to study in isolation the effect on the code base of adding type annotations.

For each pure commit, we compare the number of type errors before and after the type annotation change. We find that 81 commits introduce more errors, 34 commits reduce the type errors and 319 commits keep the same number of errors. While in most cases (319) the number of type errors remains the same, the kind of pure commit has a significant impact on incrementing the number of errors.

Finding 9: Adding type annotations can introduce new type errors, so this process should come with the usage of type checkers.

4 DISCUSSION

Implications for developers and project managers. The overall trend is that type annotations are increasingly popular (RQ 1), suggesting that developers should pick up the habit of adding annotations as they write code. Regularly adding annotations is common especially for projects with many contributors (RQ 2). Adding to benefits reported by others [12, 18, 28], our results provide empirical motivation for adding type annotations, such as the fact that more annotations help find more type errors (RQ 4) and the observation that most annotations remain in the code for a long time (RQ 3). We also pinpoint specific update patterns for individual annotations, which could help developers to avoid recurring mistakes, e.g., related to `optional` types (RQ 3). Finally, we show that developers do not need to annotate every program, because even in projects with heavy use of annotations, self-explanatory parameters and variables often remain unannotated (RQ 1).

Implications for researchers and tool builders. Even though type annotations are being used more and more, the large majority of code elements that could be annotated currently remains unannotated (RQ 1). While probably not all code in all projects needs type

annotations, we see a huge potential for techniques that automate the process of adding types into an existing code base, such as neural type prediction models [1, 23, 30, 38]. Another promising direction is to improve the integration of type checking into the development process. The fact that many commits contain type errors found by a type checker (RQ 4), but nevertheless are committed, shows that type checking currently is not yet standard. Better understanding the reasons for this phenomenon will be interesting future work.

Threats to validity. Our selection of projects is based on popularity and the projects' creation time. Another selection strategy, e.g., based on application domains, might give other results. We focus on popular projects because they overall have a higher impact and are more likely to represent serious development efforts than, e.g., small toy projects or student assignments. To study type errors, we use a single type checker, pyre, and other type checkers may give other type errors. See Rak-amnouykit et al. [39] for a discussion of the subtle differences between the type systems behind pyre and mypy. Some of our results are based on manual inspection and heuristic algorithms, which likely are imperfect. To mitigate this threat, we carefully check all results and make them available as a reference for future work. Finally, our study focuses on a single language, Python, and we cannot claim that our results will generalize to others. Comparing the evolution of type annotations across different languages will be interesting future work.

5 RELATED WORK

Studies of Type Annotations. Some prior work studies type annotations. The perhaps most closely related study is by Rak-amnouykit et al. [39], who study type annotations and type errors in a single version of popular Python projects. In contrast to this work, they do not analyze the evolution of type annotations and type errors.

Jin et al. [26] study type annotations in 17 Python projects. They find six patterns that type annotations practices follow and they find three features of Python type annotation files. Our study differs from their result, because we analyze many more projects and we focus on different kinds of research questions focusing more on single type annotations and type errors.

Khan et al. [28] study type errors in Python repositories using *mypy*. They conclude that many type defects can be avoided (15%) by simply integrating a type checker in the software development process. Then, they find that junior and senior Python developers make a similar number of errors, concluding that the experience is not always enough to avoid this kind of errors. Our study differs from this one, because we not only analyze the number of type errors, but we study the relationship and the evolution between type annotations and type errors. Moreover, our study focuses on type annotations and not only type errors.

Researchers also study type annotations in programming languages other than Python, e.g., relating static type checking in JavaScript with known bugs [12]. Bogner and Merkel [4] compare JavaScript and TypeScript focusing on code quality and readability. However, also as our findings show, they find that TypeScript does not always guarantee fewer errors. Another study suggests that the time spent adding static types to programs is helpful in impacting the overall effort required for bug resolution [53]. Finally, several studies of dynamically typed languages focus on questions complementary to ours, e.g., the use of dynamic language constructs [42], performance issues [43], and security vulnerabilities [44].

Type Prediction for Dynamically Typed Languages. Techniques for predicting type annotations in dynamically typed languages fall into three categories. First, static type inference [3, 11, 22, 25] computes types using, e.g., abstract interpretation or type constraint propagation. While sound by design, these approaches are limited by the dynamic nature of the languages like JavaScript and Python. Second, dynamic type inference [2, 41] tracks data flows during an execution of a program, which yields precise types but is limited by code coverage. Third, probabilistic type prediction propagates and combines type hints using probabilistic rules [52] or learns a machine learning from existing type annotations [1, 23, 30, 38, 40]. Our work underlines the need for such techniques and motivates future improvements.

Program Analysis for Python. Beyond type prediction, several other analyses for Python have been proposed, including techniques to find type-related bugs [51], an analysis to reveal inconsistencies between the name of a variable and the runtime values stored in it [37], and a general-purpose dynamic analysis framework [8]. These analyses are all based on dynamic analysis, which is at least partially motivated by the lack or incompleteness of type annotations in Python.

Tracking Code Elements Across Version Histories. Tracking code elements across the different commits in a project is a challenging problem due to the various ways how code may change, and because there is no universally accepted definition of when a code element remains "the same" across a change. Grund et al. [15] propose CodeShovel, which addresses this problem on the method level through an AST-based, heuristic algorithm. Ketkar et al. [27] focus on type changes in Java, using type fact graphs to represent code changes. Our algorithm for extracting type annotation changes is the first attempt at tracking annotations in a dynamically typed language.

Analyzing Code Changes. There are many approaches to analyze and study code changes. Several techniques build edit scripts on ASTs [9, 10, 20], providing an abstract representation of a change that can then be applied in different scenarios [31]. Lase generalizes multiple code changes into a single edit script [32]. Paletov et al. [36] study code changes related to crypto APIs and they extract security fixes from code histories. Weissgerber et al. [47] identify code changes that have a high chance to be refactored. Hashimoto et al. propose a technique for reducing a diff to the essence of a bug [21]. SCC [13] and DeepJIT [24] are predictive models that estimate the correlation between the insertion of a code change and introducing a bug. A related problem is to find the bug-inducing code change for a given bug report [48, 49]. To simplify studies on code changes, techniques for matching commit histories against specific queries have been proposed, e.g., Prequel [29] and DiffSearch [14]. Because type annotation changes typically affect only a single code location, using a complex representation, such as AST edit scripts, is unnecessary for our study, but we instead analyze annotation changes using custom algorithms.

Studies of Code Evolution. Software is continuously evolving and many researchers perform interesting studies. Nguyen et al. [33] study the repetitiveness of code changes in code histories, modeling a code change as a pair of AST sub-trees within a method. Gu et al. [16] analyze large project histories to study problems related to multi-thread programming. Dagenais et al. [6] study code evolution to recommend relevant changes with a high precision. Our paper contributes the first in-depth study of the evolution of type annotations in Python.

Gradual Type Checking and Type Errors. Gradual type checkers [35, 46] have developed into powerful tools for dynamically typed languages. Chen et al. build a framework to check type bugs, extracting information from source code using static analysis [5]. Dolby et al. use static analysis with types to track TensorFlow behavior and find bugs [7]. The results of our study underline the need for better integrating such tools into the development workflow.

6 CONCLUSION

This paper presents a large-scale empirical study of the characteristics and evolution of type annotations and type errors in Python. Our methodology statically analyzes individual commits of projects, extracts type annotations, combines them into histories that show the evolution of the annotations, and type checks different commits of projects. We extract 1.4 million type annotation changes from 9,655 repositories. Our results show that type annotations are clearly gaining traction, yet the large majority of code elements that could be annotated currently remains unannotated. While probably not all code in all projects needs type annotations, we see a huge potential for techniques that automate the process of adding types into an existing code base, such as neural type prediction models [1, 23, 30, 38]. Finally, many developers seem to not regularly check their code for statically detectable type errors, or if they do, commit the code despite such errors. We recommend to increase developer awareness and to better integrate gradual type checkers into the development process to alleviate this situation.

ACKNOWLEDGMENT

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *PLDI*.

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 459–472. https://doi.org/10.1145/1926385.1926437

[3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming*, Andrew P. Black (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 428–452.

[4] 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. *arXiv preprint arXiv:2203.11115* (2022).

[5] Lin Chen, Baowen Xu, Tianlin Zhou, and Xiaoyu Zhou. 2009. A Constraint Based Bug Checking Approach for Python. In *Computer Software and Applications Conference (COMPSAC)*. IEEE, 306–311.

[6] Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Trans. Softw. Eng. Methodol* 20, 4 (2011).

[7] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 1–10.

[8] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *FSE*.

[9] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[10] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering* 33, 11 (2007), 725–743.

[11] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. 2009. Profile-guided static typing for dynamic scripting languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 283–300.

[12] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To type or not to type: Quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 758–769.

[13] Emanuel Giger, Martin Pinzger, and Harald C Gall. 2011. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 83–92.

[14] Luca Di Grazia, Paul Bredl, and Michael Pradel. 2022. DiffSearch: A Scalable and Precise Search Engine for Code Changes. *CoRR* abs/2204.02787 (2022). https://doi.org/10.48550/arXiv.2204.02787 arXiv:2204.02787

[15] Felix Grund, Shaiful Alam Chowdhury, Nicholas Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *ICSE*.

[16] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. 2015. What change history tells us about thread synchronization. In *ESEC/FSE*. 426–438.

[17] Stefan Hanenberg. 2010. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 22–35.

[18] Stefan Hanenberg. 2010. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 22–35.

[19] 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* 19, 5 (2014), 1335–1382.

[20] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*. 279–288. https://doi.org/10.1109/WCRE.2008.44

[21] Masatomo Hashimoto, Akira Mori, and Tomonori Izumida. 2018. Automated patch extraction via syntax- and semantics-aware Delta debugging on source code changes. 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*. 598–609. https://doi.org/10.1145/3236024.3236047

[22] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *International Conference on Computer Aided Verification*. Springer, 12–19.

[23] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. 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*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 152–162. https://doi.org/10.1145/3236024.3236051

[24] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. 34–45. https://doi.org/10.1109/MSR.2019.00016

[25] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Symposium on Static Analysis (SAS)*. Springer, 238–255.

[26] Wuxia Jin, Dinghong Zhong, Zifan Ding, Ming Fan, and Ting Liu. 2021. Where to Start: Studying Type Annotation Practices in Python. In *ASE*.

[27] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding type changes in java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 629–641.

[28] Faizan Khan, Boqi Chen, Daniel Varro, and Shane Mcintosh. 2021. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* (2021).

[29] Julia Lawall, Quentin Lambert, and Gilles Muller. 2016. Prequel: A Patch-Like Query Language for Commit History Search. (2016).

[30] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

[31] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: generating program transformations from an example.. In *PLDI*. 329–342.

[32] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples.. In *ICSE*. 502–511.

[33] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *European Software Engineering Conference and International Symposiumon Foundations of Software Engineering (ESEC/FSE)*. ACM, 268–278.

[34] John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. 2018. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 190–201.

[35] Francisco Ortin, Jose Baltasar Garcia Perez-Schofield, and Jose Manuel Redondo. 2015. Towards a static type checker for python. In *European Conference on Object-Oriented Programming (ECOOP), Scripts to Programs Workshop, STOP*, Vol. 15. 1–2.

[36] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin T. Vechev. 2018. Inferring crypto API rules from code changes. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 450–464.

[37] Jibesh Patra and Michael Pradel. 2022. Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks. In *ICSE*.

[38] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-based Validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 209–220. https://doi.org/10.1145/3368089.3409715

[39] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. [n.d.]. Python 3 Types in the Wild: A Tale of Two Type Systems. ([n. d.]).

[40] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code".. In *Principles of Programming Languages (POPL)*. 111–124.

[41] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The Ruby Type Checker. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (Coimbra, Portugal) *(SAC '13)*. ACM, New York, NY, USA, 1565–1572. https://doi.org/10.1145/2480362.2480655

[42] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In *European Conference on Object-Oriented Programming (ECOOP)*. 52–78.

[43] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.

[44] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security Symposium*. 361–376.

[45] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie van Deursen. 2017. Why and how JavaScript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 578–589.

[46] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for python. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Laurence Tratt (Eds.). ACM, 45–56. https://doi.org/10.1145/2661088.2661101

[47] Peter Weißgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 231–240.

[48] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 262–273. https://doi.org/10.1145/2970276.2970359

[49] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23, 5 (2018), 2866–2900. https://doi.org/10.1007/s10664-017-9567-4

[50] Xinmeng Xia, Xincheng He, Yanyan Yan, Lei Xu, and Baowen Xu. 2018. An Empirical Study of Dynamic Types for Python Projects. In *International Conference on Software Analysis, Testing, and Evolution*. Springer, 85–100.

[51] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. 2016. Python predictive analysis for bug detection. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 121–132. https://doi.org/10.1145/2950290.2950357

[52] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 607–618. https://doi.org/10.1145/2950290.2950343

[53] Jie M. Zhang, Feng Li, Dan Hao, Meng Wang, Hao Tang, Lu Zhang, and Mark Harman. 2021. A Study of Bug Resolution Characteristics in Popular Programming Languages. *IEEE Trans. Software Eng.* 47, 12 (2021), 2684–2697. https://doi.org/10.1109/TSE.2019.2961897