All or Nothing: How Library Type Annotations Affect Client-Side Errors in Python

Eric Asare✉
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
eric.asare@nyu.edu

Luca Di Grazia
University of St. Gallen
St. Gallen, Switzerland
work@lucadigrazia.com

Sarah Nadi
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
sarah.nadi@nyu.edu

Abstract

Python's gradual typing promises that library-side type annotations will propagate safety benefits to downstream clients. Despite over 83% of popular libraries now containing annotations, no empirical evidence exists on whether these efforts actually reduce type errors in client code. We present the first large-scale empirical study of this downstream impact, analyzing the interaction between 13,751 actively maintained client repositories and over 4,178 distinct third-party libraries using the Pyright static type checker. Our results show that (i) 70% of client repositories contain at least one type error stemming from third-party API usage, (ii) the error distribution is heavily skewed, with a small subset of widely used libraries, i.e., `numpy`, `django`, `pandas`, and `pytorch`, accounting for a disproportionate share of the ecosystem-wide error burden, (iii) measurable benefits of reduced error risks and propagation can only be seen with near-complete annotation coverage ($\geq$ 90%), and (iv) although 60% of library-client pairs are aligned in their typing efforts, only 10% achieve the ideal scenario where highly annotated libraries meet client-side static type checking. Our findings expose a pervasive ecosystem gap: client adoption of static type checking outpaces library annotation efforts, with 19% of cases representing missed opportunities where clients use type checkers but their dependencies lack sufficient annotations. These results call for prioritizing comprehensive annotation of widely used libraries to unlock the full potential of Python's gradual typing.

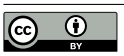
CCS Concepts

• **Software and its engineering → Software libraries and repositories**.

Keywords

Type Annotations, Third-Party Libraries, Client-Side Errors

ACM Reference Format:
Eric Asare, Luca Di Grazia, and Sarah Nadi. 2026. All or Nothing: How Library Type Annotations Affect Client-Side Errors in Python. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834434



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834434

1 Introduction

Software development heavily relies on third-party libraries, but correctly using external Application Programming Interfaces (APIs) remains a persistent challenge [5, 30]. Errors such as passing incorrect data types are a frequent source of runtime crashes and bugs [4, 19]. To mitigate these problems, Python introduced optional static typing [32]. Type annotations serve as machine-readable documentation that specifies API typing requirements, allowing static type checkers (such as Mypy [26], Pyright [24], Pyre [15], and Pytype [17]) to automatically verify client code against these requirements [9]. By leveraging these tools, developers can catch type mismatches in API calls early in the development lifecycle as static type errors, rather than discovering them as runtime failures in production [14, 20, 21, 36]. Beyond early bug detection, annotations also improve developer productivity through enhanced code readability, program comprehension, and IDE support [18, 35].

Recognizing these benefits, many Python projects have adopted type annotations over time [14], with even higher adoption within third-party libraries. A recent study [6] found that 83.39% of 11,021 popular libraries on the Python Package Index (PyPI) contain at least one annotation. Within these adopting libraries, the median proportion of annotated functions (a.k.a. *annotation coverage*) is 19.68%. Their findings demonstrate that library maintainers are adopting annotations at a higher rate than general applications to improve code clarity, linting, and static analysis for their users.

Despite this growing adoption of annotations in third-party libraries, a critical research gap remains: *it is unclear whether the promised benefits of type annotations actually transfer to the libraries' clients*. The ecosystem currently operates on an untested assumption that library-side type safety seamlessly propagates downstream. Yet no empirical evidence maps library-side typing efforts to measurable downstream benefits in client code. For example, does depending on a library whose APIs are mostly annotated actually lead to fewer type errors when using those APIs? Additionally, how well do library maintainers' annotation efforts align with their clients' adoption of static type checking?

Figure 1 illustrates how a library's type annotation directly prevents a downstream bug. In this example, the client repository inadvertently passes an incorrectly typed argument to a third-party API (`humanize`) in Figure 1b. Because the library maintainer annotated the API's parameters (Figure 1a), the client's static type checker (e.g., Pyright) immediately flags the mismatch before runtime in Figure 1c. We refer to such an error as a *LibClientError*. If the library were unannotated, this mismatch would fail silently during static analysis and manifest as a runtime crash.

In this work, we present the first large-scale empirical study investigating the downstream impact of library type annotations

```
def naturaldelta(value: dt.timedelta | float,
    months: bool = True, minimum_unit: str = "
    seconds") -> str:
                ...
```

(a) Type annotations on library's API

```
import humanize
# Passes a string instead of a timedelta/float
time_str = "today"
result = humanize.time.naturaldelta(time_str)
```

(b) Client code with a type mismatch.

```
error: Argument of type "Literal['today']"
cannot be assigned to parameter "value"
of type "timedelta | float" in function
"naturaldelta" (reportArgumentType)
```

(c) Pyright catching the type error.

Figure 1: Example of a third-party type error (*LibClientError*) caught by a client's static type checker due to library annotations.

in Python by examining whether client repositories that depend on annotated libraries experience measurable benefits. We analyze the interaction between 13,751 actively maintained client repositories and 4,178 unique third-party libraries to understand how type safety propagates. Specifically, our research questions are:

- **RQ1 How often do client developers make type errors related to third-party libraries?** We first quantify how often client repositories contain static type errors involving third-party APIs (i.e., LibClientErrors).
- **RQ2 Are certain libraries more prone to type errors?** We identify which libraries frequently result in *LibClientErrors*. This allows us to determine which dependencies are most prone to causing type-related friction for their users and for the ecosystem.
- **RQ3 Does annotation coverage of Python libraries affect the error risk experienced by their clients?** We analyze whether differences in library annotation coverage correspond to differences in client-side type error outcomes.
- **RQ4 How well aligned are library annotation coverage and client adoption of static type checking?** We investigate the alignment between library annotation coverage and client-side adoption of static type checking. This identifies "missed opportunities" where either libraries lack types or clients fail to utilize available type information.

Our results reveal that 70% of client repositories contain at least one LibClientError, with a mean of 84 and a median of 14 errors per affected repository. By analyzing the propagation and burden of these errors, we find that a small subset of widely used libraries, i.e., `numpy`, `pytorch`, `pandas`, and `django` are particularly problematic. We find that libraries with near-complete annotation coverage (≥90%) exhibit significantly lower downstream issues (e.g., error propagation and related risks) compared to libraries with lower annotation coverage, with moderate effect sizes. Interestingly, we do not observe this difference at moderate annotation coverage thresholds (e.g., 50-70%), suggesting that a measurable downstream benefit of annotation coverage is only realized when a library is almost fully annotated. Finally, although 60% of library-client pairs are aligned in their typing efforts, the ideal scenario of high annotation coverage meeting client-side type checking occurs in only 10% of cases, revealing an ecosystem gap where client adoption of type checking does not match library annotation efforts.

In summary, this paper makes the following contributions:

- **Prevalence analysis.** The first large-scale empirical study quantifying the prevalence and distribution of static type errors caused by third-party library APIs (*LibClientErrors*) across 13,751 Python repositories and 4,178 unique dependencies.
- **Coverage–safety relationship.** A systematic analysis of the relationship between library annotation coverage and client-side type safety, revealing a threshold effect where comprehensive annotation (≥90% coverage) is necessary to achieve meaningful downstream error reductions.
- **Ecosystem alignment assessment.** An evaluation of the alignment between library maintainers' annotation practices and client developers' adoption of static type checking, identifying pervasive missed opportunities within the Python typing ecosystem.

2 Background and Scoping

Type Annotations and Type Checkers in Python. Python supports optional static typing through type annotations introduced in PEP 484. Type annotations allow developers to specify the expected types of variables, function parameters, and return values without affecting runtime semantics. For example:

```
def add(x: int, y: int) -> int:
    return x + y
```

Here, `x` and `y` are annotated as `int`, and the function is declared to return an `int`. Annotations can also express container and composite types (e.g., `list[str]`, `dict[str, int]`), union types (e.g., `int | None`), and user-defined classes. These annotations are not enforced at runtime by the Python interpreter; instead, they are analyzed by external static type checkers.

Static type checkers such as MyPy [26], Pyright [24], and Pyre [15] analyze source code without executing it. Given a program and its annotations, a type checker performs static analysis to infer types, propagate constraints through assignments and function calls, and detect inconsistencies between expected and actual types. For example, passing a `str` argument to a function parameter annotated as `int` results in an error. Type checkers typically implement variations of gradual typing, allowing partially annotated codebases while inferring types for unannotated components when possible.

A prior large-scale empirical evaluation [21] reports that Pyright achieves the highest recall (74.55%) and F1 score (57.95%) among major Python type checkers, including Pyre, Pytype, and MyPy. Accordingly, we use Pyright in this study. Furthermore, Pyright provides fine-grained error categories that distinguish different classes of type inconsistencies (e.g., argument mismatches, assignment incompatibilities, and return-type violations). This granularity enables us to focus on the errors relevant to our study goals.

Third-Party Libraries and API Misuse. Third-party libraries offer APIs intended for downstream client usage. An API misuse is a violation of an implicit or explicit API usage constraint [5, 30]. Client developers using dynamically typed languages often struggle with correctly using third-party library APIs due to the dynamic nature of the language [19]. He et al. found that most of the observed errors in using Python APIs stem from its dynamic nature [19], with incorrect assumptions about the type of passed arguments representing a major burden [19]. Accordingly, when studying type errors in client projects, we focus on capturing misuses at the boundary between client code and library APIs. To achieve this, we use Pyright's type check `reportArgumentType` rule. This rule is triggered when the type of an argument provided to an API call is incompatible with the corresponding API parameter's annotated type (capturing incorrect API inputs). This allows us to analyze mismatches between the intended and actual usage of library APIs.

Type Annotation Coverage. To allow clients to use static type checkers to validate correct typing in a library's API usage, libraries must add type annotations to their functions, especially public APIs. Previous work defined *annotation coverage* [6] as the percentage of the library's functions that are fully annotated (i.e., the functions' return type and all parameters have type annotations). In our work, we use the same definition of annotation coverage to relate the library's typing efforts to downstream API usage in client code.

3 Methods

This section discusses the methods we use for our empirical study. We start by explaining how we select the target projects for the study. We then explain how we detect type errors, compute library annotation coverage, and identify type checker adoption. Figure 2 provides an overview of our research methodology, illustrating the flow from data collection to metric extraction and evaluation.

3.1 Selecting Target Projects

As subjects for our study, we select a wide range of open-source Python projects that are actively maintained and rely on third-party libraries. We gather the initial list of projects via the SEART GitHub Search Engine [12]. To ensure that we analyze non-trivial, real-world development efforts, we follow established practices in mining software repositories studies [13, 22, 25, 37]. Specifically, we apply the following inclusion criteria during our extraction: (i) Python as the primary language, (ii) at least one commit within the last 12 months to confirm active maintenance, (iii) a minimum of 100 commits, (iv) at least 100 stars, (v) a minimum of 3 contributors, and (vi) exclusion of repository forks. This query yields an initial set of 17,073 repositories. Thirteen of these repositories fail to clone because they no longer exist on GitHub. This leaves us with 17,060 repositories.

To further refine our dataset and eliminate noise, we perform an automated filtering step. We remove archived projects and repositories containing zero lines of Python code (209 repositories in total). To exclude non-software projects, such as tutorials, books, and curated lists, we adopt the filtering criteria used in previous mining studies [10, 11, 29, 34]. Specifically, we search the repository names, tags, and descriptions for the following keywords: "tutorial", "demo", "example", "book", "ebook", and "awesome-list." This excludes a further 506 repos.

Because our analysis focuses on the interaction between client code and external libraries, we require the presence of dependency configuration files to allow us to identify dependencies. Accordingly, we retain only repositories containing at least one of the common Python dependency files: `setup.py`, `requirements.txt`, or `pyproject.toml`. This further removes 1,735 repositories. Finally, since we use Pyright to detect type errors (see section 3.2), we filter out 859 repositories for which Pyright could not run. This filtering process leaves us with 13,751 remaining repositories that we use for our study.

Algorithm 1 Extraction of Library Type Errors and Alignment Metrics

Require: $\mathcal{R}$: Target dataset of filtered client repositories
Ensure: $\mathcal{E}$: Set of LibClientErrors, $\mathcal{M}$: Set of Alignment Metrics

1: $\mathcal{E} \leftarrow \emptyset$
2: $\mathcal{M} \leftarrow \emptyset$
3: **for** each repository $r \in \mathcal{R}$ **do**
4: $V_r \leftarrow CreateIsolatedEnv(r)$
5: $InstallDependencies(r, V_r)$
6: *// 2-Pass Pyright Execution for Type Errors*
7: $RunPyright(r, V_r)$ {Pass 1: Detect missing imports}
8: $InstallMissingImports(r, V_r)$
9: $E_r \leftarrow RunPyright(r, V_r)$ {Pass 2: Final analysis}
10: **for** each error $err \in E_r.reportArgumentType$ **do**
11: $f \leftarrow ExtractFullyQualifiedName(err)$
12: $\ell \leftarrow ExtractTopLevelModule(f)$
13: **if** $\ell \in V_r.ThirdPartyPackages$ **then**
14: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(r, err.arg, f, \ell, err.loc)\}$
15: **end if**
16: **end for**
17: *// Client Adoption and Alignment*
18: $has\_tc \leftarrow CheckTypeCheckerAdoption(r)$ {mypy, pyright, etc.}
19: **for** each dependency version (ℓ, v) installed in V_r **do**
20: $cov_{\ell,v} \leftarrow ComputeAnnotationCoverage(\ell, v)$
21: $is\_high\_cov \leftarrow (cov_{\ell,v} \geq 0.90)$
22: $alignment \leftarrow DetermineAlignment(is\_high\_cov, has\_tc)$
23: $\mathcal{M} \leftarrow \mathcal{M} \cup \{(r, \ell, v, cov_{\ell,v}, alignment)\}$
24: **end for**
25: **end for**
26: **return** $\mathcal{E}, \mathcal{M}$

3.2 Detecting Type Errors

As outlined in Algorithm 1, our overall extraction pipeline operates in several rule-based steps built around Pyright and structured outputs, enabling a large-scale automated study. For each client repository, we first create an isolated virtual environment (Line 4) and install all dependencies listed in the repository's dependency files (e.g., `requirements.txt`) (Line 5). We track the complete set of installed dependencies and their versions using `pip freeze`.

We then run Pyright on the client repository to detect type errors. To reduce errors caused by missing dependencies, our Pyright

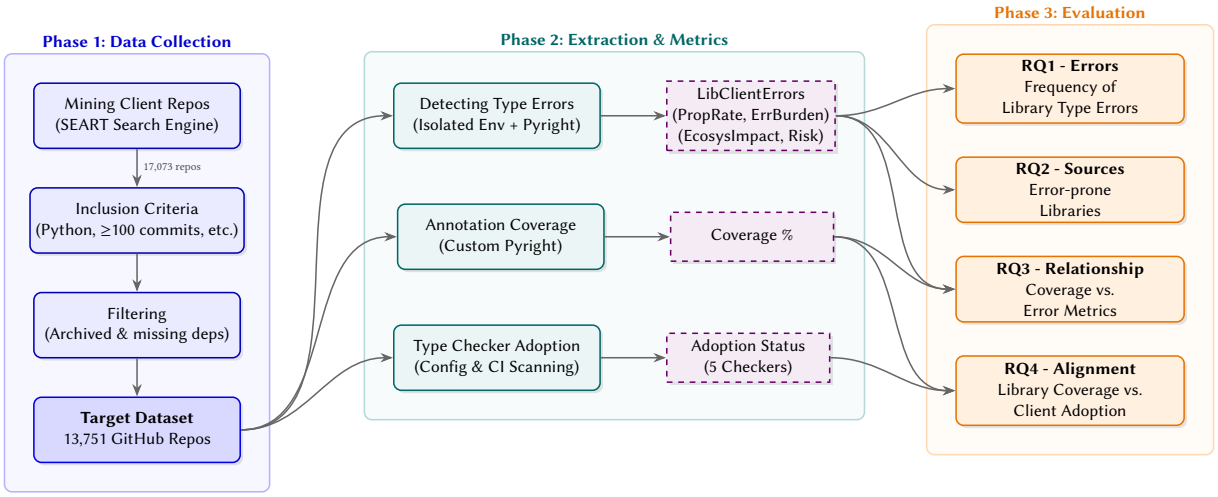


Figure 2: Overview of the research methodology, illustrating the three main phases: (1) Data Collection to curate the target repository dataset, (2) Extraction of type errors, annotation coverage, and type checker adoption metrics, and (3) Evaluation mapping the extracted metrics to the research questions (RQ1–RQ4).

execution is in two passes. First, we run Pyright and then collect all `reportMissingImports` errors (Line 7). We automatically parse these missing imports and install the corresponding dependencies into the isolated analysis environment (Line 8). In some cases, the distribution name differs from the import name (e.g., `beautifulsoup4` is imported as `bs4`), which may prevent us from resolving a Pyright-reported missing import during the re-install. However, this does not cause us to miss the library entirely unless it is absent from the dependency files of every repository in our dataset, which is highly unlikely. For example, `beautifulsoup4` is declared as a dependency by 2,831 client repositories in our dataset.

We then re-execute Pyright to extract actual type errors that are not related to missing imports (Line 9). Our analysis focuses on the `reportArgumentType` rule provided by Pyright, which flags argument type incompatibilities during the evaluation of call expressions.

To attribute errors to specific libraries, we modify Pyright to emit error messages containing the fully qualified name of the target function rather than only the function name that was shown in Figure 1c. With this modification, we can simply parse the structured output of the diagnostic message to extract the fully qualified name and identify the library (Lines 10 - 16).

For example, consider the following new error message format:

```
Argument of type "Literal['today']"
    cannot be assigned to parameter "
    value" of type "datetime.timedelta
    | builtins.float" in function "
    humanize.time.naturaldelta"
```

We extract the fully qualified function name by matching the quoted identifier following the substring `"in function "` (Line 11). In this example, the extracted identifier is `humanize.time.naturaldelta`. We then attribute the error to the corresponding library using the top-level module name (Line 12), which is the first segment of the dot-separated identifier, i.e., `humanize` in our example.

To distinguish third-party dependencies from project-internal modules, we compare the extracted top-level module name against the set of installed packages in the analysis environment (Lines 13-15). Because each repository is analyzed in an isolated environment whose installed packages are recorded, we can reliably determine whether the module belongs to a third-party dependency. We classify error messages whose top-level module corresponds to an installed third-party package as *Library Client Errors* (LibClientError). LibClientErrors are therefore third-party related argument-type mismatches under Pyright, not necessarily a crashing runtime bug. We treat all others as internal errors. Accordingly, we obtain a set of LibClientError defined as follows.

Definition 3.1 (Library Client Error). A library client error (LibClientError) is a tuple $e = (r, a, f, \ell, loc)$, where r is the client repository in which the error occurs, a is the mismatched argument provided by the client, f is the target function or method being invoked, ℓ is the third-party library to which f belongs, and loc is the code location (file path and line number) of the call expression.

To quantify the frequency and effect of library related type errors, we define the following metrics. Let $clients(\ell)$ denote the set of repositories whose analysis environment contains library ℓ, and let E_r denote the set of LibClientErrors observed in repository r.

Definition 3.2 (Affected Clients). The *affected clients* of library ℓ, denoted $AffectedClients(\ell)$, is the number of client repositories in which at least one LibClientError involving ℓ is observed:

$$AffectedClients(\ell) = |\{r \in clients(\ell) \mid e_{r,\ell} > 0\}|$$

where $e_{r,\ell} \in E_r$ denotes the number of LibClientErrors involving ℓ in repository r.

Definition 3.3 (Propagation Rate). The *propagation rate* of library ℓ, denoted $PropRate(\ell)$, is the proportion of its client repositories that are affected:

$$PropRate(\ell) = \frac{|AffectedClients(\ell)|}{|clients(\ell)|}.$$

Definition 3.4 (Error Burden). The *error burden* of library ℓ in repository $r \in AffectedClients(\ell)$ is the proportion of LibClientErrors in r that involve ℓ:

$$ErrBurden(\ell, r) = \frac{e_{r,\ell}}{E_r},$$

Accordingly, the aggregate error burden of a library is the median of its error burden per affected client repository:

$$ErrBurden(\ell) = \text{median}_{r \in AffectedClients(\ell)}\ ErrBurden(\ell, r).$$

Overall, $PropRate(\ell)$ captures how frequently a client of ℓ experiences type errors because of its use (i.e., likelihood of a problem), while $ErrBurden(\ell, r)$ measures how much of a given repository's third-party type errors is attributable to ℓ (i.e., severity of the problem).

Definition 3.5 (Risk). The *risk* level of using a library ℓ, denoted $Risk(\ell)$, is the product of its propagation rate and error burden:

$$Risk(\ell) = PropRate(\ell) * ErrBurden(\ell)$$

Definition 3.6 (EcosystemImpact). The *ecosystem impact* of a library ℓ, denoted $EcosysImpact(\ell)$, is the product of the number of its affected clients and median error burden across affected clients:

$$EcosysImpact(\ell) = AffectedClients(\ell) * ErrBurden(\ell)$$

In other words, $Risk(\ell)$ tells us how risky it is for a client to use this library, measured through both likelihood of having errors and the severity/burden of these errors. On the other hand, $EcosysImpact(\ell)$ estimates the total aggregate error burden contributed by a library across all affected clients across the ecosystem. We use this estimate as opposed to just total errors across all affected repos to avoid one client having a lot of errors incorrectly skew the results. Such an estimate allows us to rank libraries by their overall ecosystem effect.

3.3 Extracting Library Type Annotation Coverage

To relate errors in client code to the typing efforts of library maintainers, we compute the type annotation coverage *within* the third-party libraries used by clients in our dataset. We use Asare and Nadi's tooling for calculating annotation coverage [6]. They provide a modified version of Pyright that extracts both public and private functions and methods in the library's source code and determines whether they contain explicit type annotations. Note that we calculate annotation coverage for each library version that appears in the dependencies of our target projects, given the following definition:

Definition 3.7 (Library Annotation Coverage). Given a version v of a third-party library ℓ, its type annotation coverage is defined as the ratio of fully typed functions and methods to the total number of functions and methods in that version:

$$cov_{\ell,v} = \frac{|\{f \in \ell_v \mid f \text{ is annotated}\}|}{|\{f \in \ell_v\}|}$$

f is considered annotated if all its arguments and return types are annotated as in the example described in background [6].

The clients in our dataset could be using different versions of the same library, with each version having a different annotation coverage. While we can consider each (ℓ, v) as a separate data point in our analysis, differences in the number of times a specific version appears in the dataset can inflate the data with repeated data points that look similar, especially since previous work showed that most libraries have constant coverage [6]. Accordingly, we calculate a single annotation coverage metric for each library as the median of annotation coverage of all its versions that appear in the client repositories.

$$cov(\ell) = \text{median}_{v \in versions(\ell)}\ cov(\ell, r).$$

3.4 Identifying Client Type Checker Adoption

To study how clients align with library annotation efforts, we detect the usage of static type checkers within a repository. We consider five widely used Python type checkers: `mypy`, `pyright`, `pyre`, `pytype`, and `pyrefly` [27] and define adoption as follows.

Definition 3.8 (Type Checker Adoption). A client repository r is considered to have adopted a type checker if it satisfies at least one of the following conditions:

(1) $\exists$ a configuration file $c \in r$ specific to the given type checker. More precisely:
   - for `mypy`: $c \in$ {`mypy.ini`, `.mypy.ini`},
   - for `pyright`: $c =$ `pyrightconfig.json`,
   - for `pyre`: $c \in$ {`.pyre_configuration`, `.pyre_configuration.local`},
   - for `pytype`: $c \in$ {`pytype.cfg`, `pytype.toml`},
   - for `pyrefly`: $c =$ `pyrefly.toml`.

(2) $\exists$ a Github Action CI/CD workflow file $w \in R$ containing an executable `run` block that invokes the type checker.

(3) $\exists$ a tool-specific declaration section s within the `pyproject.toml` file of r (e.g., `[tool.mypy]`).

By systematically scanning the repository structure and configuration manifests for these indicators, we partition the studied repositories into those that actively enforce static typing and those that do not. This enables an analysis of how library maintainers' type-safety efforts align with those of their clients.

3.5 Measuring Alignment Between Library-Client Pair

We define *library–client alignment* as the consistency between a library's annotation coverage and whether the client repository uses a static type checker. Alignment captures whether the availability of type annotations in a library matches the client's use of static type checking. Practically, when a library provides extensive annotations, client repositories can only benefit from these annotations if they use a type checker. However, a client that uses a type checker but depends on libraries with limited annotations experiences reduced

benefits in terms of type checking library interactions. Accordingly, we consider a library-client pair as *aligned* in two situations:

- The library has low annotation coverage and the client repository does not use a static type checker.
- The library has high annotation coverage and the client repository uses a static type checker.

Conversely, a pair is considered *misaligned* (a missed opportunity) in two situations:

- *Client missed opportunity*: the library has high annotation coverage but the client repository does not use a static type checker.
- *Library missed opportunity*: the library has low annotation coverage while the client repository uses a static type checker.

We experiment with different annotation coverage thresholds in RQ3 and use the result to define *high annotation coverage* in RQ4. Low coverage corresponds to annotation coverage below this threshold. Finally, for each library-client pair, we determine whether the client repository r uses a static type checker, as described in Section 3.4.

3.6 Implementation

We conduct all experiments on a high-performance computing (HPC) cluster. We develop custom scripts to clone and process GitHub repositories, automating data collection, type analysis, and aggregation. We parallelize experiments across compute nodes using the SLURM job scheduler, with each job utilizing up to 120 CPU cores and 16 GB of RAM. To ensure a consistent execution environment, we use a Singularity container based on CentOS Linux 8.2.2004, which includes Git 2.18.4 for cloning. We use Python versions ≥ 3.9 for most of our automated scripts, and Pyright version 1.1.408, along with all required dependencies. The total cloned repositories required approximately 1,600 GB of storage.

4 Results

We now present the results of each of the four research questions.

4.1 RQ1: How often do client developers make type errors related to third-party libraries?

Across the 13,751 repositories in our dataset, we find that 13,608 (99%) repositories contain at least one static type error, regardless of whether the errors stem from third-party library usage. The number of statically detected type errors per repository ranges from 0 to 489,577, with a median of 316 errors. This tells us that type errors are generally prevalent.

When we specifically look at type errors related to third-party library usage, Figure 3 shows that 9,583 (70%) of the 13,751 repos have LibClientErrors (i.e., have at least one argument type error related to the usage of a third-party library). Across these 9,583 repos, the number of LibClientErrors ranges from 1 to 14,067 with a mean of 84 errors and a median of 14 errors per repo. These results show that the third-party API boundary is a pervasive source of type-related friction: the majority of actively maintained Python repositories contain type mismatches while calling a library that could be caught by static analysis.

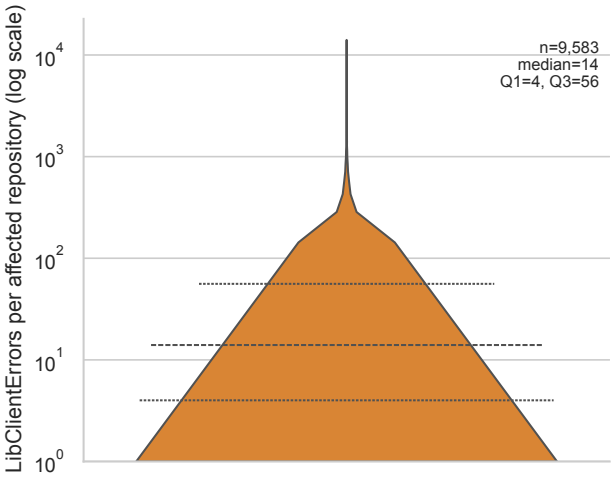


Figure 3: LibClientErrors across affected repos

RQ1 Summary: 70% of the client repositories in our dataset have at least one LibClientError with a mean of 84 and median of 14 LibClientErrors per affected repo.

4.2 RQ2: Are certain libraries more prone to type errors?

Our goal is to understand if some libraries are more problematic than others when it comes to type errors. There are 28,333 dependencies that appear across our 13,751 analyzed repositories. We find that, on average, a dependency is used by 35.03 clients (min: 1, max: 12,072, IQR: 1–4). However, 14,557 (51%) of these dependencies appear in only one repository (i.e., have a single client), while 24,155 (85%) appear in fewer than ten repositories. However, 4,178 (15%) libraries have at least 10 clients. Accordingly, we focus any dependency level analysis on these 4,178 dependencies, allowing us to still include dependencies below the average number of clients while avoiding a long tail of outliers with almost no clients.

We find that the majority (59%) of the 4,178 libraries have no client-side type errors related to their usage (i.e., have PropRate=0). Among the remaining 1,720 libraries with non-zero propagation, the mean PropRate is 12%, with a median of 7% and an interquartile range of 3%–17%. For these libraries, the mean ErrBurden is 24%, with a median of 9% and an interquartile range of 3%–38%. This means that, on average, only 12% of a library's client base will suffer from type errors due to its usage. When errors do occur for such a library, they represent almost a quarter (24%) of the third-party library type errors a client developer needs to deal with.

Figure 4 shows a scatter plot of PropRate vs. ErrBurden for each of the 1,720 dependencies, color-coded by the number of *affected clients* for each library. We can clearly see that many of the libraries with high ErrBurden (top half of the graph) are in grey, indicating that they have only one affected client. Considering libraries with only one (or even few) affected clients is misleading when analyzing problematic libraries. Conversely, most of the libraries with high propagation rates (right side of the graph) have more

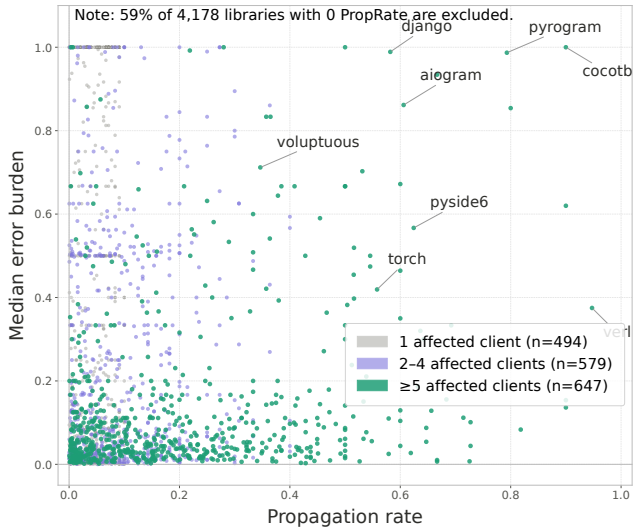


Figure 4: Median error burden vs. propagation rate per library; color coding indicates number of affected clients

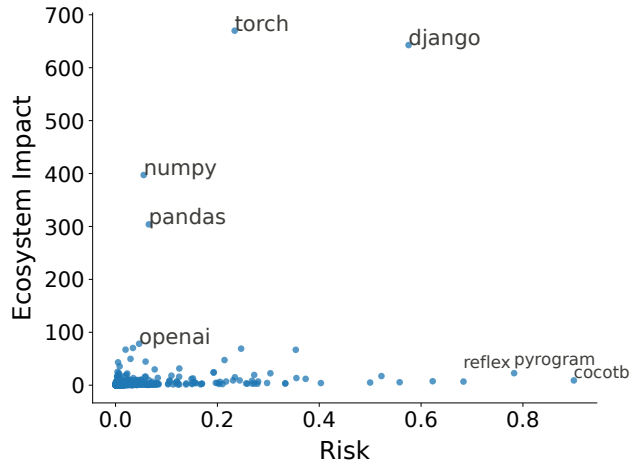


Figure 5: Risk vs. Ecosystem Impact for 647 Libraries with >= 5 affected clients

than 5 affected clients. Accordingly, to achieve a more meaningful analysis of problematic libraries, we turn to the $Risk(\ell)$ and $EcosysImpact(\ell)$ metrics that provide a more comprehensive analysis, analyzing these scores only for the 647 libraries with at least five affected clients.

Figure 5 shows a scatter plot of EcosysImpact vs. Risk for these 647 libraries. We can see that most libraries lie in the low risk, low impact bottom left quadrant of the graph. Libraries such as reflex, pyrogram, cocotb are in the bottom right quadrant where using them has a high risk of introducing type errors in the client code; however, they have a low ecosystem impact indicating that these high number of errors appear in a small number of repos. At the top half of the graph, we see numpy, pytorch, pandas, and django.

Table 1: Risk and Impact Summary for Top 10 dependencies in terms of number of Clients

Library	# Clients	# Affected Clients	Risk	Ecosystem Impact
typing_extensions	12,072	38	0.000133	1.604
packaging	11,121	119	0.000124	1.384
pygments	9,905	20	0.000070	0.692
tomli	9,456	5	0.000011	0.102
urllib3	9,185	117	0.000389	3.573
requests	8,795	605	0.004914	43.214
six	8,618	40	0.000225	1.938
python-dateutil	7,543	118	0.000533	4.018
jinja2	7,379	131	0.000934	6.895
numpy	7,186	2694	0.055272	397.188

Numpy, pytorch, and pandas have very high ecosystem impact but a low risk. This means that while these libraries do not necessarily result in a high error burden per repository, they do result in errors in many of their clients, increasing their overall error impact on the ecosystem. The most notable problematic library is django, which has both a high ecosystem impact and a high risk, suggesting that it is both widely used and highly likely to result in many type errors in its clients.

We also examine the data in terms of libraries with the most number of clients, however also limiting it to those with at least five affected clients. Table 1 summarizes the top 10 libraries in our data set. We find that well-known libraries such as urllib3 and requests are commonly used without causing type errors (i.e., they have both a low risk and high ecosystem impact).

RQ2 Summary: The majority of libraries have low risk and low ecosystem impact in terms of type errors. Four particularly impactful libraries stand out: `numpy`, `pytorch`, `pandas`, and `django`, with `django` having both high risk and high ecosystem impact.

4.3 RQ3: Does annotation coverage of Python libraries affect the error risk experienced by their clients?

In RQ3, we investigate whether the extent of a library's type annotations affects the type errors in its clients. We expect that a library with a higher proportion of its APIs being annotated (i.e., higher annotation coverage) would result in less LibClientErrors with its usage. Hence low *propRate*, *ErrBurden*, and *Risk* levels. To assess the relationship between coverage and client outcomes, we partition library versions into low- and high-coverage groups using progressively stricter thresholds. We compare distributions in the low (x) and high (y) coverage groups using the Mann–Whitney U test and report rank-biserial correlation (RBC) and common language effect size (CLES) for effect size interpretations. RBC represents the gap between how often evidence supports a result versus how often it contradicts it. It has a value from -1 to 1, where positive values suggest that x>y (i.e., low coverage results in more type error impact) while a negative value suggests the opposite (i.e., that low coverage

results in less type error impact). CLES represents the proportion of pairs where x>y.

To ensure meaningful error comparisons, we restrict our analysis to the 4,178 libraries with more than 10 clients. However, we could obtain the annotation coverage data for only 1,640 of these libraries. The remaining libraries had various data collection failures or inconsistent metadata, including cases where `total_functions` was reported as zero despite evidence that the library defined functions. Accordingly, we run the analysis for the 1,640. We find that annotation coverage for these libraries ranges from 0%– 100% with Q3 at 71%.

Table 2 shows the relationship between annotation coverage and the three metrics (*ErrBurden*, *propRate*, and *Risk*) at different thresholds defining low vs. high coverage, for 1,640 libraries at least 10 clients. We observe no statistically significant differences at the baseline threshold (Q3 = 71%), with negligible effect sizes across all outcomes (RBC = 0.02-0.05, CLES = 0.51-0.53). As the threshold increases, the differences become statistically significant and effect sizes increase consistently. At stricter thresholds, lower-coverage libraries tend to exhibit higher propagation rates, greater error burden, and increased client risk. This is reflected in positive RBC values and CLES values exceeding 0.57 across all outcomes, indicating that lower-coverage libraries are more likely to exhibit worse client-level effects.

Given that RQ2 showed that many of the libraries are rarely used, we repeat the analysis on a subset of 519 libraries with at least 100 clients to assess whether removing less widely used libraries affects the observed relationships. We show the results in Table 3. Restricting to libraries with at least 100 clients yields a similar pattern, though with larger effect sizes. Statistically significant differences emerge at Q3+10% for Risk, and across all three metrics at Q3+20%. Effect sizes increase monotonically from RBC = 0.12 at Q3 = 71% to RBC = 0.33 at high annotation coverage ($Q3 + 20\%$), with CLES increasing from 0.56 to 0.66, larger than those observed for the broader library set. This suggests that among more widely used libraries, the association between lower coverage and worse client-level effects is more pronounced.

RQ3 Summary: Libraries with higher annotation coverage are more likely to have less client-side type errors, with the strongest effects observed at high coverage levels (≥90%), where library usage is significantly less likely to result in type errors in a client.

4.4 RQ4: How well aligned are library annotation coverage and client adoption of static type checking?

In RQ4, we investigate the alignment between library annotation coverage and client-side adoption of static type checking. Specifically, we examine whether highly annotated libraries are used by clients that employ type checkers, and conversely, whether clients using type checkers depend on well-annotated libraries. We identify "missed opportunities", where libraries used in type-checking clients lack sufficient type annotations or clients fail to utilize available type information by not employing type checkers.

To analyze this relationship, we consider the 1,640 libraries with annotation data and all of their 13,426 corresponding clients. Since a repository may use multiple libraries, our data has 566,877 library-repository pairs. We assign each library-repository pair to one of four alignment categories based on the library's median annotation coverage and the client's type-checking usage. We use a threshold of 90% annotation coverage to distinguish between low and high coverage libraries, motivated by the findings of RQ3.

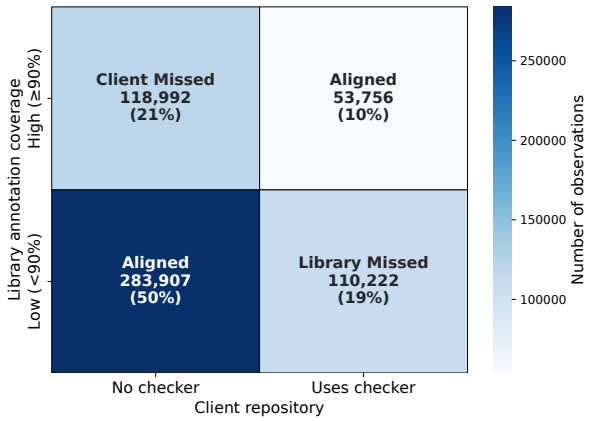


Figure 6: Alignment between a library's annotation coverage and type checker usage in its clients. Each observation represents one of 566,877 library-client pairs. The darker the shade of blue, the higher the number of observations.

Figure 6 shows these four categories as a 2×2 alignment matrix with a heat map. We find that 60% of observations are aligned, while 40% correspond to misaligned cases. The largest share of observations (50%) lies in the aligned low-coverage/no-checker quadrant, where libraries have low annotation coverage and client repositories do not use static type checking, indicating lack of interest/effort in type checking from both library and client sides. In contrast, only 10% of observations fall into the ideal aligned high-coverage/has-checker quadrant, where libraries have high annotation coverage and client repositories use static type checking.

Misalignment arises in two forms. In 21% of observations, libraries provide high annotation coverage but client repositories do not use static type checking (client-side missed opportunities). Conversely, in 19% of cases, repositories use static type checking while depending on a library that has low annotation coverage (library-side missed opportunities).

To further interpret these results, we repeat a similar analysis on the library level. Out of the 1,640 libraries considered, 228 (14%) of these have high coverage. Out of these 228 high-coverage libraries, a median of 70% of their respective clients *do not* use type checkers. This further highlights client missed opportunities.

On the other hand, if we focus only on the client level, we find that 3,628 (27%) of the considered clients use type checkers. For these 3,628 repos, 100% depend on at least one library with low annotation. Worse, on average, 59% of each of these repos' dependencies have low annotation (i.e., only an average of 41% of its dependencies have high coverage it can benefit from).

Table 2: Mann–Whitney U results comparing client-level effects between a total of 1,640 low- and high-coverage libraries with >= 10 clients across annotation coverage thresholds. Holm correction is applied; statistically significant results shown in bold. RBC is Rank-Biserial Correlation and CLES is Common Language Effect Size.

	ErrBurden			PropRate			Risk		
Threshold	**p-value**	**RBC**	**CLES**	**p-value**	**RBC**	**CLES**	**p-value**	**RBC**	**CLES**
Q3: < 71% vs ≥ 71%	5.83×10^{-1}	0.02	0.51	3.01×10^{-1}	0.05	0.53	4.11×10^{-1}	0.04	0.52
Q3+10%: < 81% vs ≥ 81%	1.65×10^{-1}	0.07	0.54	$\mathbf{1.11 \times 10^{-2}}$	**0.11**	**0.56**	$\mathbf{1.39 \times 10^{-2}}$	**0.11**	**0.55**
Q3+20%: < 91% vs ≥ 91%	$\mathbf{1.14 \times 10^{-3}}$	**0.16**	**0.58**	$\mathbf{8.68 \times 10^{-6}}$	**0.21**	**0.60**	$\mathbf{1.14 \times 10^{-5}}$	**0.20**	**0.60**

Table 3: Mann–Whitney U results comparing client-level effects between a total of 1,640 low- and high-coverage libraries with >= 100 clients across annotation coverage thresholds. Holm correction applied; statistically significant results shown in bold. RBC is Rank-Biserial Correlation and CLES is Common Language Effect Size.

	ErrBurden			PropRate			Risk		
Threshold	**p-value**	**RBC**	**CLES**	**p-value**	**RBC**	**CLES**	**p-value**	**RBC**	**CLES**
Q3: < 71% vs ≥ 71%	7.45×10^{-2}	0.12	0.56	8.05×10^{-2}	0.09	0.55	7.45×10^{-2}	0.13	0.56
Q3+10%: < 81% vs ≥ 81%	7.45×10^{-2}	0.14	0.57	7.45×10^{-2}	0.14	0.57	$\mathbf{1.38 \times 10^{-2}}$	**0.17**	**0.59**
Q3+20%: < 91% vs ≥ 91%	$\mathbf{3.66 \times 10^{-4}}$	**0.27**	**0.63**	$\mathbf{7.11 \times 10^{-5}}$	**0.29**	**0.65**	$\mathbf{5.02 \times 10^{-6}}$	**0.33**	**0.66**

RQ4 Summary: While the typing efforts of 60% of library-client pairs are aligned, only 10% correspond to the ideal scenario where highly annotated libraries are used by type-checked client repositories. The 40% misaligned cases are split almost evenly between client missed opportunities (21%) and library missed opportunities (19%). On average, a repository using a type checker has 41% of its dependencies with high coverage while a library with high coverage has, on average, 30% of its clients use type checkers.

5 Discussion

Our findings show a complex ecosystem where the adoption of static typing is highly asymmetric, creating friction points that manifest as third-party argument type errors. We discuss the implications of these results for library maintainers, client developers, and researchers.

5.1 The Ecosystem Typing Gap

RQ1 establishes that argument type errors involving third-party libraries are pervasive: 70% of repositories contain at least one LibClientError; these are preventable errors that persist in actively maintained codebases. RQ2 reveals that this distribution is heavily skewed: a small number of widely used libraries such as `numpy`, `django`, and `pandas` contribute a significant share of the ecosystem-wide error burden. This concentration suggests that targeted annotation improvements in a small set of high-impact libraries could yield substantial ecosystem-wide benefits.

Recent industry efforts support this view: organizations such as Meta and Quansight have invested in improving type annotations across widely used Python libraries, including the scientific Python stack (e.g., `pandas`) [23].

RQ3 shows that at moderate coverage thresholds, the relationship between annotation coverage and reduced error propagation is weak. However, beyond 90% coverage, the effect becomes both statistically significant and practically meaningful (CLES up to 0.65 among widely used libraries).

Relating the findings of RQ2 and RQ3 reveals an important ecosystem pattern. The libraries that contribute disproportionately to client-side type errors, including `numpy`, `pytorch`, `pandas`, and `django`, also have relatively low median annotation coverage of 17.2%, 26.4%, 14.2%, and 3.5%, respectively. None of these libraries approach the high annotation coverage levels (≥90%) that RQ3 associates with significantly lower client-side type error risk. This *threshold effect* challenges the assumption that any typing effort is beneficial and instead suggests that library maintainers should aim for comprehensive coverage to realize meaningful downstream returns.

The alignment analysis in RQ4 contextualizes why this threshold effect matters in practice. The largest share of library-client observations (50%) falls into a "mutually untyped" state where the potential benefits of gradual typing are unrealized on both sides. More critically, 19% of observations represent library missed opportunities, where clients actively use static type checking but depend on libraries with low annotation coverage. On average, a repository that uses a type checker has only 41% of its dependencies with high coverage. These clients are investing in type safety but cannot fully realize its benefits due to upstream annotation gaps. Worse, a library that invested in high coverage has, on average, only 30% of its clients benefit from this effort while the remainder do not employ type checkers.

5.2 Implications for Practitioners

For Library Maintainers: Our results suggest that partial type annotations offer limited downstream value. Because significant reductions in error propagation only emerge at high coverage thresholds (≥90%), maintainers of widely used libraries should prioritize comprehensive typing of their API surfaces. This is particularly

relevant for libraries like `django`, which exhibits both high risk and high ecosystem impact. Django's dynamic nature makes straightforward type annotations insufficient for many core components such as models, querysets, and forms, as reflected in a long-running pull request debating type adoption within the project [3]. The community's reliance on a third-party stubs package [1] further illustrates the challenge, as annotation effort has been displaced outside the core project rather than integrated directly. Since manually adding type annotations can be time-consuming, library maintainers can instead generate them using type recommendation tools such as Stray [31] or PyAnnGen [18], which improves upon existing approaches.

For Client Developers: Developers adopting static type checking cannot assume their tools will catch third-party mismatches out of the box due to the prevalent alignment gap. When depending on an unannotated or partially annotated library, practitioners should turn to available stub files such as those provided by Typeshed [2] or community-maintained packages like `django-stubs` [1], which provide type information without requiring changes to the upstream library. Until widely used libraries achieve high annotation coverage, such stubs represent the most practical path to meaningful static analysis at dependency boundaries.

5.3 Implications for Researchers

Targeted Automated Repair and Inference: The 19% library-missed opportunities highlight a gap where client repositories adopt type checking, but the libraries they depend on lack sufficient annotations. This suggests an opportunity for automated type inference and stub generation techniques to provide type information without requiring intervention from library maintainers. Overall, the misalignment identified in RQ4 reflects a structural imbalance: client adoption of type checking is progressing faster than the availability of well-annotated libraries, motivating research into scalable approaches for automatically generating and maintaining type information for third-party libraries.

6 Threats to Validity

Construct Validity. We identify client libraries via `pip freeze`, which may include installed but unused packages, providing a conservative (lower-bound) estimate of propagation rates. Our analysis focuses on a single Pyright rule (`reportArgumentType`), one of the most directly relevant for library API misuse [19]. We do not capture other error categories (e.g., assignment incompatibilities), making our results represent a lower bound of the number of potential type errors. Some developers may locally use static type checkers in their IDEs or on the command line; we cannot capture such usage. Instead, we focus on repo-level enforced checker usage where we consider both build and CI configurations. The presence of a configuration file or tool declaration does not necessarily imply that the type checker is executed regularly in practice, but it does provide evidence of intended or declared adoption for measurement purposes.

Internal Validity. We used Pyright given its highest recall and F1 score among major Python type checkers [21]. Other tools (e.g., `mypy`) might yield different distributions, though we expect overall trends to hold as argument type checking is fundamental to all checkers. We mitigate confounding factors in RQ3 by analyzing at multiple thresholds and filtering by popularity, observing consistent trends.

External Validity. Our study focuses on open-source Python GitHub projects. The observed dynamics may not generalize to strictly typed ecosystems or closed-source codebases. However, our findings are relevant to the growing number of gradually typed languages (e.g., TypeScript, Ruby with Sorbet) that face similar library-client typing challenges. Our client selection criteria focus on actively maintained projects, which we consider a strength: our results characterize the most actively developed segment of the Python ecosystem.

7 Related Work

The evolution and adoption of type annotations in Python. The integration of type hints in Python has evolved from a theoretical addition to a widespread practice. Earlier studies in 2019 [28] reported low annotation adoption rates where only 3.78% of the studied repositories contained partial annotations. Later studies in 2021 showed that adoption rose to 7% [14] but with less than 10% annotation coverage. Most recently, Asare and Nadi focused specifically on third-party libraries (as opposed to any Python repository) and found that 83% of 11,021 actively used packages on the Python Package Index have adopted at least one type annotation [6], with a median coverage of 20%. Their longitudinal analysis identified stable or increasing annotation coverage over time with minimal abandonment. In contrast to these studies focusing on annotation adoption, our work empirically examines client-side type mismatches across different levels of annotation coverage in third-party libraries.

The prevalence of type-related defects within projects. Empirical studies consistently highlight argument type errors as a significant category of software failures. Khan et al. demonstrated that approximately 15% of all type-related defects in Python could be completely avoided by integrating modern static type checkers [20]. Furthermore, their analysis revealed that both junior and senior developers introduce a statistically similar volume of type errors, underscoring that automated static analysis is a mandatory safeguard against increasingly complex application states. Chen et al. [8] show that bad typing practices significantly correlate with bugs. Unlike these studies that examine the general prevalence of type-related defects within projects, no prior work has empirically examined how upstream library annotations reduce these mismatches in client code.

The effectiveness of static type checkers. The tooling landscape for Python type checking has rapidly evolved to support massive enterprise codebases. Xu et al. [36] showed that adding type annotations significantly improved bug detection. Chow et al. [9] introduced PyTy, an automated program repair approach specifically targeted at statically detectable type errors in Python. Their model, trained via cross-lingual transfer learning, successfully addresses real-world errors using type-checker for the error localization and for validating the fix suggested by the model. While prior work highlights the effectiveness of static type checkers, our work bridges a gap by investigating their role in enforcing type constraints in client

repositories and examining how this role aligns with typing efforts on the library side.

Third party libraries and APIs. The boundary between client code and third-party libraries is particularly susceptible to integration failures, commonly referred to as API misuse [30]. Wei et al [33] studied API misuse of Python deep learning libraries, finding that 51% of the deep learning misuses are related to API methods while 35% involved API parameters. Galappaththi et al. [16] studied a broader set of data centric Python libraries and found that 43% of the misuses involved API methods, and 51% involved API parameters. More broadly, He et al [19] studied real-world API misuses in Python programs and found that the dynamic nature of Python accounts for 25% of API misuses related to passed arguments in Python programs. By focusing on the call site via `reportArgumentType`, we target the interface where external library dependencies most frequently collide with client implementations. While prior work studies a range of API misuses, both related and unrelated to typing, our work focuses specifically on typing-related API usage issues. In particular, we examine the downstream ecosystem impact of third-party libraries and how their typing practices align with client-side type errors.

8 Conclusion

This paper presented the first large-scale empirical study on the downstream impact of library type annotations in the Python ecosystem. Analyzing 13,751 client repositories and 4,178 third-party libraries, we find that 70% of repositories contain type errors from third-party API usage. Our analysis reveals a threshold effect: partial annotations provide limited downstream benefits, but almost comprehensive coverage (≥90%) is associated with significantly reduced error propagation. Our ecosystem alignment analysis further exposes a pervasive gap: only 10% of library-client pairs achieve the ideal scenario of high annotation coverage meeting client-side type checking, while 19% represent actionable missed opportunities where clients use type checkers but their dependencies lack sufficient annotations. These results carry a clear message: to unlock the full potential of gradual typing, the community must prioritize comprehensive API annotation in widely used libraries. Automated type inference approaches, guided by ecosystem-level impact metrics such as those proposed in this work, can help bridge the current gap and build a more type-safe software supply chain.

Data Availability Statement

All data and scripts used in this study are available in our replication package on Figshare[7].

Acknowledgments

This research was carried out using the High Performance Computing resources at New York University Abu Dhabi. Luca Di Grazia is supported by the Great Minds Fellowship of the University of St. Gallen. Gen AI tools (ChatGPT, Claude, Gemini, GitHub Co-pilot) were used in copy editing, for scripts, for data analysis, figure creation, and improving readability of the reproducibility instructions in our artifact.

References

[1] [n. d.]. django-stubs: PEP-484 stubs for Django. https://github.com/typeddjango/django-stubs.
[2] [n. d.]. typeshed: Collection of library stubs for Python. https://github.com/python/typeshed.
[3] 2023. DEP 0065: Type Annotations. https://github.com/django/deps/pull/65.
[4] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: a benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 464–467. https://doi.org/10.1145/2901739.2903506
[5] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1170–1188. https://doi.org/10.1109/TSE.2018.2827384
[6] Eric Asare and Sarah Nadi. 2026. How do third-party Python libraries use type annotations?. In *23rd International Mining Software Repositories Conference (MSR 2026)*. https://doi.org/10.1145/3793302.3793337
[7] Eric Asare, Sarah Nadi, and Luca Digrazia. 2026. Replication Package for All or Nothing: How Library Type Annotations Affect Client-Side Errors in Python. (8 2026). https://doi.org/10.6084/m9.figshare.31892497.v2
[8] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. An Empirical Study on Dynamic Typing Related Practices in Python Systems. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 83–93. https://doi.org/10.1145/3387904.3389253
[9] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 87, 13 pages. https://doi.org/10.1145/3597503.3639184
[10] Jailton Coelho, Marco Tulio Valente, Luciana L. Silva, and André Hora. 2018. Why we engage in FLOSS: answers from core developers. In *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering* (Gothenburg, Sweden) *(CHASE '18)*. Association for Computing Machinery, New York, NY, USA, 114–121. https://doi.org/10.1145/3195836.3195848
[11] Jailton Coelho, Marco Tulio Valente, Luciana L. Silva, and Emad Shihab. 2018. Identifying unmaintained projects in github. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 15, 10 pages. https://doi.org/10.1145/3239235.3240501
[12] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 560–564. https://doi.org/10.1109/MSR52588.2021.00074
[13] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 235–245. https://doi.org/10.1109/ICSME55016.2022.00029
[14] 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3540250.3549114
[15] Facebook. 2018. Pyre: Static Type Checker for Python. https://pyre-check.org/.
[16] Akalanka Galappaththi, Sarah Nadi, and Christoph Treude. 2024. An Empirical Study of API Misuses of Data-Centric Libraries. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain) *(ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/3674805.3686685
[17] Google. 2012. Pytype: Static Type Analyzer for Python. https://github.com/google/pytype. Infers types without requiring annotations; support until Python 3.12.
[18] 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 Trans. Softw. Eng. Methodol.* 33, 5, Article 123 (June 2024), 38 pages. https://doi.org/10.1145/3652153
[19] Xincheng He, Xiaojin Liu, Lei Xu, and Baowen Xu. 2023. How Dynamic Features Affect API Usages? An Empirical Study of API Misuses in Python Programs. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 522–533. https://doi.org/10.1109/SANER56733.2023.00055
[20] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. 2022. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* 48, 8 (2022), 3145–3158. https://doi.org/10.1109/TSE.2021.3082068
[21] Xinrong Lin, Baojian Hua, Yang Wang, and Zhizhong Pan. 2023. Towards a Large-Scale Empirical Study of Python Static Type Annotations. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 414–425. https://doi.org/10.1109/SANER56733.2023.00046

[22] Petr Maj, Stefanie Muroya, Konrad Siek, Luca Di_Grazia, and Jan Vitek. 2024. The fault in our stars: Designing reproducible large-scale code analysis experiments. (2024). https://doi.org/10.4230/LIPIcs.ECOOP.2024.27

[23] Meta Engineering. 2025. Enhancing the Python ecosystem with type checking and free threading. https://engineering.fb.com/2025/05/05/developer-tools/enhancing-the-python-ecosystem-with-type-checking-and-free-threading/ Accessed: 2026-03-26.

[24] Microsoft. 2019. Pyright: Static Type Checker for Python. https://github.com/microsoft/pyright.

[25] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[26] Mypy Developers. 2012. Mypy: Optional Static Typing for Python. http://mypy-lang.org/.

[27] Wonseok Oh and Hakjoo Oh. 2024. Towards effective static type-error detection for python. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1808–1820. https://doi.org/10.1145/3691620.3695545

[28] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/3426422.3426981

[29] Xiaoning Ren, Yuhang Ye, Xiongfei Wu, Yueming Wu, and Yinxing Xue. 2025. Demystifying the Evolution of Neural Networks with BOM Analysis: Insights from a Large-Scale Study of 55,997 GitHub Repositories. arXiv:2509.20010 [cs.SE] https://arxiv.org/abs/2509.20010

[30] Michael Schlichtig, Steffen Sassalla, Krishna Narasimhan, and Eric Bodden. 2022. FUM - A Framework for API Usage constraint and Misuse Classification. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 673–684. https://doi.org/10.1109/SANER53432.2022.00085

[31] Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. 2023. Static Type Recommendation for Python. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 98, 13 pages. https://doi.org/10.1145/3551349.3561150

[32] Guido van Rossum. 2014. *The Python Language Reference*. Python Software Foundation. https://docs.python.org/3/reference/

[33] Moshi Wei, Nima Shiri Harzevili, Yuekai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2024. Demystifying and Detecting Misuses of Deep Learning APIs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 201, 12 pages. https://doi.org/10.1145/3597503.3639177

[34] Xiaoya Xia, Shengyu Zhao, Xinran Zhang, Zehua Lou, Wei Wang, and Fenglin Bi. 2023. Understanding the Archived Projects on GitHub. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 13–24. https://doi.org/10.1109/SANER56733.2023.00012

[35] Wentian Xie, Sufan Zhang, and Ziyuan Wang. 2025. Do Type Annotations Help Students Improve Python Program Development Efficiency?. In *2025 25th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. 594–600. https://doi.org/10.1109/QRS-C65679.2025.00078

[36] Wenjie Xu, Lin Chen, Chenghao Su, Yimeng Guo, Yanhui Li, Yuming Zhou, and Baowen Xu. 2023. How Well Static Type Checkers Work with Gradual Typing? A Case Study on Python. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. 242–253. https://doi.org/10.1109/ICPC58990.2023.00039

[37] Weiqin Zou, Weiqiang Zhang, Xin Xia, Reid Holmes, and Zhenyu Chen. 2019. Branch Use in Practice: A Large-Scale Empirical Study of 2,923 Projects on GitHub. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. 306–317. https://doi.org/10.1109/QRS.2019.00047