Co-evolution of Types and Dependencies: Towards Repository-Level Type Inference for Python Code

SHUO SUN, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
SHIXIN ZHANG, Institute of Software, Chinese Academy of Sciences, China
JIWEI YAN*, Institute of Software, Chinese Academy of Sciences, China
JUN YAN*, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China
JIAN ZHANG, Institute of Software, Chinese Academy of Sciences, China and University of Chinese Academy of Sciences, China

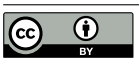
Python's dynamic typing mechanism, while promoting flexibility, is a significant source of runtime type errors that plague large-scale software, which inspires the automatic type inference techniques. Existing type inference tools have achieved advances in type inference within isolated code snippets. However, repository-level type inference remains a significant challenge, primarily due to the complex inter-procedural dependencies that are difficult to model and resolve. To fill this gap, we present PyTIR, a novel approach based on LLMs that achieves repository-level type inference through the co-evolution of types and dependencies. PyTIR constructs an Entity Dependency Graph (EDG) to model the objects and type dependencies across the repository. During the inference process, it iteratively refines types and dependencies in EDG for accurate type inference. Our key innovations are: (1) an EDG model designed to capture repository-level type dependencies; (2) an iterative type inference approach where types and dependencies co-evolve in each iteration; and (3) a type-checker-in-the-loop strategy that validates and corrects inferences on-the-fly, thereby reducing error propagation. When evaluated on 12 complex Python repositories, PyTIR significantly outperformed prior works, achieving a *TypeSim* score of 0.89 and a *TypeExact* score of 0.84, representing a 27% and 40% relative improvement over the strongest baseline. More importantly, PyTIR removed new type errors introduced by the tool by 92.7%. This demonstrates a significant leap towards automated, reliable type annotation for real-world Python development.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Software maintenance tools**.

Additional Key Words and Phrases: Type Inference, Entity Dependency Graph, Repository-level, Large Language Models

*Corresponding authors

Authors' Contact Information: Shuo Sun, sunshuo20@otcaix.iscas.ac.cn, Institute of Software, Chinese Academy of Sciences, Technology Center of Software Engineering, Beijing, China and University of Chinese Academy of Sciences, Beijing, China; Shixin Zhang, zhangshixin@otcaix.iscas.ac.cn, Institute of Software, Chinese Academy of Sciences, Technology Center of Software Engineering, Beijing, China; Jiwei Yan, yanjiwei@otcaix.iscas.ac.cn, Institute of Software, Chinese Academy of Sciences, Technology Center of Software Engineering, Beijing, China; Jun Yan, yanjun@ios.ac.cn, Institute of Software, Chinese Academy of Sciences, Key Laboratory of Systems Software and Technology Center of Software Engineering, Beijing, China and University of Chinese Academy of Sciences, Beijing, China; Jian Zhang, zj@ios.ac.cn, Institute of Software, Chinese Academy of Sciences, Key Laboratory of Systems Software, Beijing, China and University of Chinese Academy of Sciences, Beijing, China.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE075
https://doi.org/10.1145/3797103

ACM Reference Format:
Shuo Sun, Shixin Zhang, Jiwei Yan, Jun Yan, and Jian Zhang. 2026. Co-evolution of Types and Dependencies: Towards Repository-Level Type Inference for Python Code. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE075 (July 2026), 21 pages. https://doi.org/10.1145/3797103

1 Introduction

Python's *dynamic typing* provides substantial flexibility and development speed, making it especially advantageous for rapidly building data processing scripts, a key factor in its widespread adoption [8, 17]. However, as Python has evolved from a scripting language into a primary development choice for large-scale repositories [1, 33, 44], the flexibility introduces potential risks. In particular, the absence of compile-time type checking can create latent issues that may remain undetected until runtime, potentially impacting system reliability. To mitigate this issue, PEP 484 introduced a standardized syntax for *type annotations* [10] in Python, enabling static type checkers, linters, and IDEs to detect potential errors before execution. Nevertheless, writing type annotations manually can be labor-intensive for developers. A previous study [33] revealed a notably low usage rate, finding that only 2,678 out of 70,000 analyzed repositories had adopted type annotations. This observation underscores the pressing need for automatic type inference techniques to alleviate the burden of manual annotation.

In recent years, many studies have explored automatic type inference techniques for Python code, encompassing both *static-analysis-based (SA-based)* [30, 41] and *learning-based* [31, 36] approaches. On the one hand, *SA-based* approaches leverage pre-defined inference rules, which enable efficient inference with high accuracy in restricted scenarios. On the other hand, *learning-based* approaches, ranging from deep learning models to Large Language Models (LLMs), offer greater flexibility by exploiting code comprehension capabilities and Python-specific syntactic knowledge, which can achieve high accuracy when sufficient code context is available. For both types, the empirical evaluations on standardized benchmarks, primarily consisting of isolated code snippets, demonstrate the effectiveness. For example, on the widely used *ManyTypes4Py* benchmark [28], existing methods achieve an accuracy of up to 86.2% [36], indicating the potential of current techniques in practical type inference tasks. However, when applied to real-world Python repositories, which often involve complex dependencies among Python objects, existing approaches face significant challenges. As illustrated in our preliminary study (see motivating examples in Section 2.2), both *SA-based* and *learning-based* approaches, as well as native LLMs, frequently fail to infer correct types in the presence of complex dependencies, particularly inter-procedural dependencies, which are commonly encountered in real-world codebases. This limitation underscores a persistent gap between current research and its applicability to real-world Python repositories.

To fill this gap, we aim to propose a *practical* and *precise* **repository-level Python type inference** approach. Leveraging the superior semantic understanding capabilities of LLMs, we adopt LLMs as the core inference engine of our approach. Based on our investigation, we have identified three key challenges to our approach.

Challenge 1: Identifying and resolving complex *inter-procedural type dependencies*. In real-world Python repositories, objects are interconnected through both intra-procedural and inter-procedural type dependencies. In contrast to intra-procedural type dependencies, where all relevant variables exist within a local context, resolving inter-procedural type dependencies presents greater challenges. The objects involved in inter-procedural dependencies are often dispersed across various modules within the repository, making them difficult to locate. Moreover, Python's dynamic binding mechanism further complicates their identification. However, these inter-procedural dependencies frequently provide critical type inference information. Therefore, a

key challenge lies in accurately identifying and extracting these dependencies, as well as effectively incorporating them into the type inference process.

Challenge 2: Extracting compact yet highly relevant *type-inference contexts* for LLMs. On code understanding and analysis tasks, LLMs have already demonstrated remarkable capabilities. However, even with increasingly large context windows, it is still impractical to provide an entire code repository as input. Moreover, supplying excessive information at once may distract the model's attention and negatively affect inference quality. Thus, it is essential to construct *type-inference contexts* that are both compact and highly relevant to the target object. The difficulty lies in that naive context selection strategies, such as those based solely on distance, risk discarding critical information, which in turn reduces inference precision. Thus, a key challenge lies in abstracting and constructing proper contextual information that enables LLM-based analysis to obtain sufficient knowledge.

Challenge 3: Preventing *incorrect types* from propagating along dependency chains. Due to the stochastic nature of LLMs and potential hallucinations, type inference results may contain inaccuracies. Without timely correction, an incorrectly inferred type at any point in a dependency chain can propagate to subsequent dependencies, adversely affecting downstream type inference. Thus, a key challenge lies in designing real-time error detection and corrective feedback mechanisms capable of identifying and rectifying type inconsistencies before these inaccuracies.

To address these challenges, we propose **PyTIR** (**Py**thon **T**ype **I**nference on **R**epository-Level), a novel approach for practical and precise repository-level Python type inference. To address **Challenge 1**, PyTIR introduces a data structure called the **Entity Dependency Graph (EDG)**, which captures essential type dependencies. We further propose an iterative approach for repository-level EDG construction. The core idea is *Co-Evolution of Types and Dependencies*. In each iteration, the dependencies are updated based on the latest inferred type information, which in turn refines the EDG model and guides subsequent type inference. To address **Challenge 2**, PyTIR transforms the EDG into a directed acyclic graph and performs analysis following a topological ordering. After reorganization, PyTIR infers nodes with fewer dependencies first, and summarizes their type information as contextual input for dependent nodes. Moreover, nodes that mutually depend on each other are treated as a cluster, i.e., their contexts are combined, and PyTIR infers their types simultaneously. Finally, to address **Challenge 3**, PyTIR leverages a static type checker to detect potential type conflicts within the repository after each inference iteration. When a conflict is identified, a heuristic strategy is applied to locate the erroneous type annotations and fix them.

We evaluate the effectiveness of PyTIR on 12 large-scale real-world Python repositories. The evaluation measures both the accuracy of the generated type annotations and the number of type errors introduced by the type annotations. In terms of type accuracy, PyTIR outperforms all baseline methods, achieving a top score of 0.89 on *TypeSim* metric and 0.84 on *TypeExact* metric, corresponding to improvements of 27% and 40% over the previous state-of-the-art (SOTA) methods. With respect to introduced type errors, PyTIR reduces 92.7% of introduced type errors compared to the SOTA methods, resulting in an average of only 46.17 errors per repository.

To conclude, the contributions of this paper are as follows:

- We design a novel data structure, the Entity Dependency Graph (EDG), to model type dependencies for repository-level Python type inference.
- We propose an automated type inference approach for Python repositories based on the key idea, *Co-Evolution of Types and Dependencies*, achieving high accuracy while reducing type errors introduced.

- We implement our approach into an open-source tool PyTIR; we also provide a high-quality dataset comprising 12 unannotated Python repositories and their corresponding type-annotated versions, which are free of type errors and can serve as benchmarks for further evaluation.

2 Background and Motivating Example

In this section, we first describe the various type dependencies and then illustrate the challenges of repository-level Python type inference through motivating examples.

2.1 Type Dependencies in Python

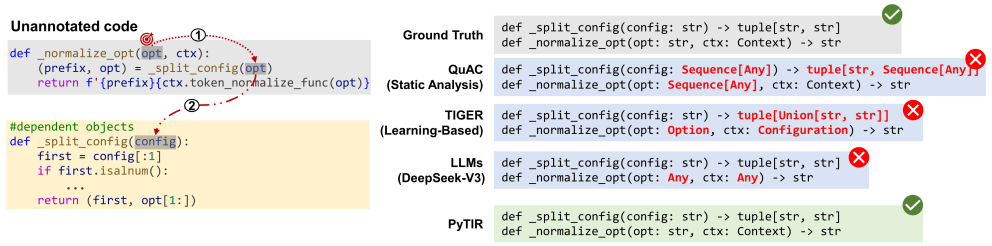
The core insight of repository-level type inference is that the types of program entities are not independent but are constrained by a complex web of relationships. Accurately modeling these dependencies is crucial for propagating type information and achieving high inference accuracy. As pointed out by prior work [30], type dependencies in Python exist between *variable occurrences* rather than *variables* themselves. It is due to Python's dynamic typing nature, where a variable can be rebound to values of different types during runtime. In this paper, we call a variable occurrence o_1 *dependent* on another variable occurrence o_2, denoted as $o_1 \rightarrow o_2$, if the type of o_2 is a prerequisite for determining the type of o_1. The common type dependencies can be categorized as follows:

- *Usage-Definition Dependency*: In Python, a variable can be reassigned to a value of a different type, and the type of a variable at a usage occurrence is determined by its most recent preceding definition occurrence. Therefore, the usage occurrence of a variable is *dependent* on its most recent preceding definition occurrence.
- *Parameter-Usage Dependency*: Due to the principle of duck typing, the set of valid types for a function parameter is constrained by how that parameter is used within the function body. Therefore, a parameter occurrence is *dependent* on all its usage occurrences within the function.
- *Argument-Parameter Dependency*: The type of an argument in a function call must be compatible with the type of the corresponding parameter. Therefore, a call-site argument occurrence is *dependent* on the function's parameter occurrence.
- *Attribute-Instance Dependency*: The type of an attribute access, such as `instance.attr`, is determined by the type of the `instance` object. Therefore, the attribute occurrence is *dependent* on the instance occurrence.
- *Semantic Dependency*: This category covers dependencies derived from the operational semantics of the code, rather than explicit syntactic links. For instance, the expression `x+y` creates a semantic dependency between `x` and `y`, as each variable constrains the possible types of the other through the semantics of the '+' operator.

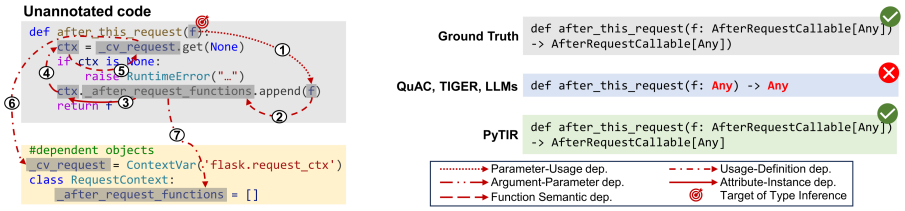
Furthermore, these dependencies can be categorized based on their scope. *Intra-procedural dependencies* are those resolved within the scope of a single function. In contrast, *inter-procedural dependencies* span across function boundaries. Resolving inter-procedural dependencies is more challenging as it requires tracking type flows across function calls, often necessitating a whole-program analysis approach.

2.2 Motivating Example

In the following, we use two real-world examples to illustrate our motivation. Fig. 1a and Fig. 1b show two code snippets collected from the widely used Python repositories *Click* [34] and



(a) Code Snippet with *Simple* Inter-Procedural Dependencies



(b) Code Snippet with *Complex* Inter-Procedural Dependencies

Fig. 1. Motivating Examples and Their Type Inference Results

Flask [35]. When applying SOTA SA-based approaches (*QuAC* [41]), learning-based approaches (*TIGER* [36]), and modern LLMs (*DeepSeek-V3 [24]*, *DeepSeek-R1* [12],*GPT-4o* [16]) for type inference, all methods fail to produce correct type annotations. The primary reason for this phenomenon lies in the various types of dependencies present in real-world repositories. By further analysis, we reveal that these approaches consistently overlook three critical aspects: (1) accurately identifying and extracting inter-procedural dependencies, (2) timely correcting erroneous type inferences due to dependency missing, and (3) iteratively co-evolving dependencies and types. Furthermore, we will provide a detailed comparison of the type inference results produced by different methods.

(1) Dependency Identification and Extraction. Fig. 1a illustrates the type inference process of the parameter `opt` in function `_normalize_opt` by tracing type dependencies. To determine the type of the parameter `opt`, the analysis must first examine its usages within the function body. As indicated by dependency ①, a key usage of `opt` is as an argument in the call to the function `_split_config`. It establishes an intra-procedural dependency between the parameter declaration and its application at the call site. Subsequently, the type constraints imposed on the argument `opt` are propagated from the corresponding parameter `config` in the callee function `_split_config`. This inter-procedural dependency, denoted by ②, demonstrates that inferring the type of the argument `opt` is contingent upon the type of the parameter `config`. The complete type inference for `opt` in `_normalize_opt` requires resolving this dependency chain, which necessitates analyzing the callee function `_split_config`. However, neither *TIGER* nor modern *LLMs* captured the inter-procedural dependency ②, consequently failing to infer the correct type for parameter `opt` in function `_normal_opt`.

To address this challenge, PyTIR designs a data structure EDG to model type dependencies and employs an iterative algorithm to extract them. As shown in Fig. 1a, PyTIR correctly identifies the inter-procedural dependency and infers the correct type for `_normalize_opt`.

(2) Error Detection and Correction. Although QuAC correctly identified the inter-procedural dependency ② in Fig.1a, it still failed to infer the correct type of `opt`. This incorrect inference stems from the wrong type of `config` in the function `_split_config`, on which `opt` depends. This error type then propagates through the dependency chain ①-②, causing further error types.

To overcome this challenge, PyTIR incorporates a static type checker to identify and correct erroneous type annotations immediately after generation, preventing further error propagation through dependency chains. Therefore, once PyTIR also generates a wrong parameter type in the function `_split_config`, PyTIR can detect the resulting type conflicts and fix the wrong type according to the error message from the static type checker.

(3) Co-evolution of Types and Dependencies. Moreover, while the demonstrated call dependency in Fig.1a is relatively straightforward, real-world repositories also involve far more intricate dependencies, which significantly hinder the accuracy of type inference. For example, Fig. 1b shows a more complex dependency chain ①-⑦ to infer the type of parameter `f` in function `after_this_request`. This chain reveals that inferring the type of `f` is contingent upon determining the type of `ctx._after_request_functions` (dependencies ①-②). Therefore, the usage-definition dependency ⑦ between the usage occurrence and definition occurrence of `ctx._after_request_functions` is crucial for inferring the type of `f`. However, due to Python's dynamic nature, without the specific type of `ctx`, the object that `ctx._after_request_functions` refers to cannot be statically resolved. Consequently, existing type inference methods QuAC, TIGER, and native LLMs all failed to identify dependency ⑦ and are thus unable to infer the type of `f`.

To resolve this challenge, PyTIR employs a co-evolution process where the dependencies and types mutually refine each other. Specifically, PyTIR first infered the type for the local variable `ctx` using dependences ④⑤⑥ in the incomplete dependency chain ①-⑥. Leveraging this newly acquired type information, it then identified the previously missing dependency ⑦, thereby completing the entire dependency chain. Finally, with the complete chain (①-⑦) established, PyTIR successfully infers the precise type of `f`.

3 Repository-Level Python Type Inference Approach

This section introduces our solution for "*how to perform practical and precise repository-level Python type inference*".

3.1 Approach Overview

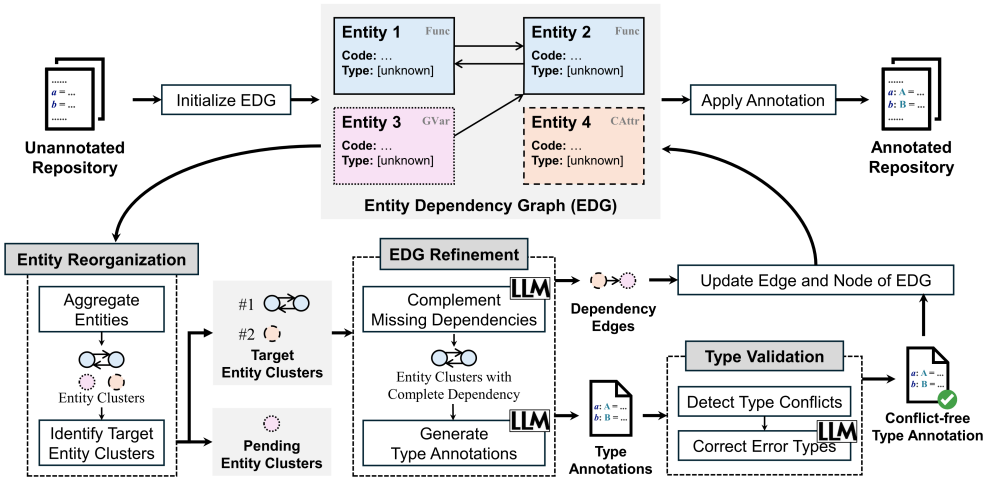


Fig. 2. Overview of PyTIR 's Workflow

Fig. 2 illustrates the workflow of PyTIR for repository-level type inference. Given a Python repository with incomplete type annotations (i.e., type hints), PyTIR constructs an Entity

Dependency Graph (EDG) to model both the objects requiring annotation and type dependencies in the repository. This graph is subsequently refined through an iterative process and ultimately applied to the original repository to produce a fully type-annotated version. The refinement iteration consists of three major phases.

(1) Entity Reorganization. To overcome the input size limitation of LLMs, PyTIR reorganizes the complete EDG into entity clusters, which serve as separate type-inference contexts provided to the LLM. Correspondingly, the dependency edges in the EDG are also remanaged into dependencies between entity clusters. Then, PyTIR selects the target entity clusters based on the topological order of these dependencies, resulting in a sequence of entity clusters to be analyzed.

(2) EDG Refinement. In this phase, PyTIR analyzes the target entity clusters to refine the dependencies and types in the EDG. PyTIR first checks whether the dependencies in the EDG are sufficient for type inference for each entity cluster, and identifies the lacking dependencies. If an entity cluster's dependencies are complete and all dependent entities are annotated, type inference will be performed on this entity cluster, with the help of LLMs.

(3) Type Validation. To timely detect type errors and prevent their propagation along dependency chains, PyTIR checks the generated type annotations with a static type checker. Once a type conflict is detected, PyTIR leverages a backtracking algorithm to search for an appropriate type to resolve the conflict. If PyTIR fails to find a suitable type, it replaces the generated type annotation with '`Any`', sacrificing some type precision to ensure that no type conflicts exist in the generated type annotations.

These three phases will be performed iteratively. In each iteration, the missing edges identified in the *EDG Refinement* Phase and the conflict-free type annotations generated in the *Type Validation* Phase are both updated into the EDG. When the type annotations for all nodes in the EDG have been inferred, the iteration terminates. PyTIR then applies the type annotations from the EDG to the repository, ultimately producing a fully annotated repository.

3.2 Design of Entity Dependency Graph

As stated in previous sections, the intra- and inter-procedural type dependencies are crucial for repository-level type inference. Thanks to their remarkable capabilities in code comprehension, LLMs can effectively identify intra-procedural type dependencies when provided with the source code of a single function. To extend this reasoning capability to encompass inter-procedural dependencies that span multiple functions, classes, and files, PyTIR transforms the entire repository into a structured, global representation, the Entity Dependency Graph (EDG). The EDG is a directed graph represented by a 2-tuple $G = \langle V, E \rangle$, where V is the set of *entities*, representing the objects that possess repository-level visibility, and $E \subseteq V \times V$ is the set of *inter-entity dependencies*, which model the inter-procedural type dependencies in the repository. In the following, we provide a comprehensive elaboration on *entities* and *inter-entity dependencies*, and detail the construction methodology of the EDG.

3.2.1 Entities. To model the inter-procedural type dependencies in the repository, PyTIR creates nodes in the EDG for objects whose types have cross-functional or global significance, which we call **Entities**. The entities we consider are as follows.

- **Variable Entities** include global variables defined at the module level and class attributes defined within classes.
- **Function Entities** include functions defined at the module, and class methods defined with classes. Specifically, the types of their parameters and return values will be annotated.
- **Class Entities** include all user-defined classes. These classes need no type annotation but can provide essential context for the type inference of other entities.

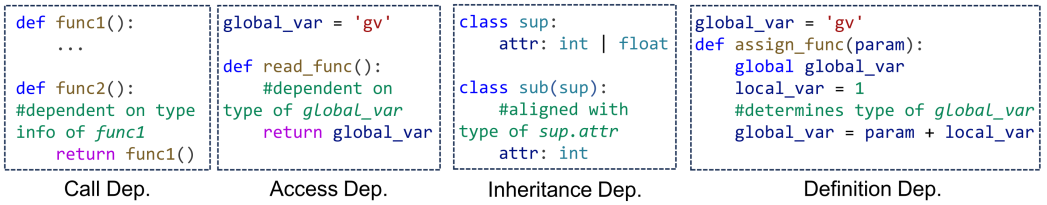


Fig. 3. Examples of Dependency Patterns in PyTIR

PyTIR does not explicitly target the inference of types for local variables within functions or for nested functions, which are typically confined to a local context. Besides, we aggregate both the parameters and the return value of a function into a cohesive *function entity*. The omission of these objects allows our analysis to focus computational resources on the dependencies that matter most for repository-level reasoning.

3.2.2 Inter-Entity Dependencies. Based on the concept of entities, we lift type dependencies from the level of variable occurrences to the level of entities. Formally, an entity E_1 *depends* on another entity E_2 if and only if there exists one variable occurrence o_1 within the syntactic scope of E_1 and another occurrence o_2 within E_2, such that o_1 is dependent on o_2.

However, precisely tracking all such dependencies via deep inter-procedural static analysis is computationally expensive and often intractable, especially in a dynamic language like Python. To address this, PyTIR employs a lightweight, pattern-based approach to efficiently identify critical inter-entity dependencies, as exemplified in Fig. 3. We define four primary dependency patterns:

- **Call Dependency (Variable/Function → Function)**. If a function entity f is called during the definition of another variable or function entity e, PyTIR establishes a dependency edge $e \rightarrow f$. The type of f's return value influences the type of the variable in e that receives this value. Moreover, the types of arguments passed from e to f should be aligned with the types of f's parameters.
- **Access Dependency (Variable/Function → Variable)**. An entity e is dependent on a variable entity v if e accesses the value of v. The type of v therefore constrains the type of the variable within e that holds the accessed value.
- **Inheritance Dependency (Class → Class)**. This dependency arises from object-oriented inheritance. A subclass C_{sub} is dependent on its parent class C_{super}. The type of a method or attribute in C_{sub} must be compatible with its declaration in C_{super}.
- **Definition Dependency (Variable → Function)**. A variable entity v is dependent on a function entity f if f assigns a value to v. The type of the value being assigned within f, i.e., the right-hand side of the assignment, constrains the type of v.

The inaccuracies introduced by our pattern-based extraction are manageable. First, superfluous edges do not significantly impede the inference capabilities of the LLM, as its attention mechanism allows it to selectively focus on the most relevant information during type resolution. Second, missing dependencies are designed to be progressively identified and incorporated during the subsequent EDG refinement phase.

3.2.3 EDG Construction. PyTIR employs static analysis on Abstract Syntax Trees (ASTs) to systematically extract entities and inter-entity dependencies from the input repository and construct the initial EDG. Specifically, PyTIR identifies entities based on the type of AST node. Furthermore, PyTIR analyzes each statement in the definition code to build dependency edges in the EDG. PyTIR determines the category of dependency, according to the pattern defined in

Sec. 3.2.2, based on the AST node type and scope, and utilizes a static analysis tool PyAnalyzer [22] to locate objects accessed within statements.

To illustrate the construction process of EDG, we revisit the motivating example of a type inference task with complex constraints, as depicted in Fig. 1b. This code snippet involves four entities: the function entity `after_this_request`, the variable entities `_cv_request` and `RequestContext._after_request_function`, and the class entity `RequestContext`. Among the two inter-procedural dependencies ⑥ and ⑦, the usage-definition dependency, labeled as ⑥, is successfully identified by matching the *Access Dependency* pattern. The complex dependency ⑦ cannot be resolved at this stage and will be progressively incorporated in subsequent iterations.

3.3 Phase I: Entity Reorganization

To construct a compact and highly relevant context for type inference, PyTIR leverages the type annotations of dependent entities, rather than their full definition code or deeper dependencies. Processing these dependencies naturally calls for a topological ordering of the EDG. However, mutual dependencies create cycles, making a standard topological sort impossible. PyTIR overcomes this by systematically identifying and collapsing these cycles into entity clusters. This core transformation reduces the EDG to a Directed Acyclic Graph (DAG) of entity clusters, enabling PyTIR to finally process them in a valid topological order for inference.

3.3.1 Aggregating Entities. PyTIR transforms the EDG into a DAG by aggregating mutually dependent entities into one entity cluster. These mutually dependent entities manifest as directed cycles within the EDG. Crucially, since cycles may share overlapping vertices, each maximal strongly connected component (SCC) represents a set of mutually dependent entities where any pair exhibits cyclic dependency. Therefore, PyTIR leverages the APIs provided by the python library *networkx* [13] to collapses such SCCs into atomic Entity Clusters, transforming the EDG into a DAG of clusters that satisfy partial ordering constraints.

Moreover, constrained by LLM context window limitations, the size of an entity cluster must be limited. PyTIR enforces an entity cluster cardinality bound of five entities by selectively removing dependency edges. To mitigate the potential loss of inference accuracy caused by edge removal, PyTIR leverages a greedy algorithm to remove dependencies inside the cluster and decompose it into small ones. Specifically, PyTIR iteratively removes dependency edges until the entity cluster is partitioned into multiple clusters that satisfy the cardinality constraint. In each iteration, PyTIR identifies an edge whose removal decomposes this SCC into multiple strongly connected components and proceeds to remove that edge. This pragmatic trade-off marginally reduces inference accuracy while preserving critical dependency relationships, maintaining computational feasibility.

3.3.2 Identifying Target Entity Clusters. As described above, PyTIR leverages the type annotations of dependent entities to supply essential dependent type information. Therefore, in each iteration, only entity clusters where all dependent entities have been annotated can serve as the target for type inference, allowing us to ignore other entity clusters to conserve computational resources. Consequently, after partitioning the EDG into entity clusters, PyTIR removes all entities that contain type annotations from the generated DAG and extracts the entity clusters with in-degree zero from the remaining graph as the objectives for type inference in the current iteration. Since the DAG is acyclic, entity clusters with in-degree zero are guaranteed to exist in the residual graph, thus ensuring that the target entity clusters can be reliably extracted in every iteration.

3.4 Phase II: EDG Refinement

In this phase, PyTIR leverages LLMs to refine both the dependency edges and type annotations of entities. The underlying idea is that, since LLMs are capable of inferring correct types when provided with sufficient contextual information, they can also identify the informational gaps that hinder the completion of the inference process. Based on this idea, after constructing the type-inference context for each target entity cluster, PyTIR sequentially performs dependency completion and type inference using this context, thereby achieving a co-evolution of type-related dependencies and type annotations.

3.4.1 Identifying Missing Dependencies. PyTIR leverages LLMs to identify the lacking edges in the EDG. Specifically, PyTIR constructs the prompt to identify the lacking dependencies for each entity cluster based on its type-inference context, including its definition code and dependencies. The primary function of these prompts is to instruct the LLM to assess whether the current type-inference context is adequate. Furthermore, they guide the LLM in pinpointing any absent dependencies that may hinder the completion of the inference process.

Consider the function entity `after_this_request` in Fig. 1b. With the type-inference context containing its definition code and the type of dependent global variable `_cv_request`, the LLM can determine that the reference of variable `_cv_request` is of type `ContextVar[RequestContext]`. Leveraging its knowledge of the Python standard library, particularly the `ContextVar` class, the LLM can further deduce that the local variable `ctx` is of type `RequestContext`. Based on these insights, the LLM identifies a critical gap: the type of `RequestContext._after_request_functions` remains unspecified in the type-inference context. Consequently, PyTIR identifies the absence of the inter-procedural dependency ⑦, which is crucial for resolving the type-inference task.

3.4.2 Generating Type Annotations. Since the missing dependencies are identified based on the latent EDG, the entity clusters that are detected to have missing dependencies may contain additional missing dependencies, which can only be identified after the EDG is updated. Therefore, PyTIR performs type inference exclusively on the entity clusters without missing dependencies. For each entity cluster with complete dependency, PyTIR constructs the prompt by utilizing the type-inference context of the entity cluster and a predefined prompt template, and queries LLMs to generate the type annotations.

3.5 Phase III: Type Validation

The final phase of the iteration is dedicated to ensuring the correctness and consistency of the generated type annotations. It operates through a two-step process: first, detecting type conflicts using a static type checker, and second, employing a backtracing approach to correct the identified errors. The ultimate objective is to produce a set of *conflict-free* type annotations that can be successfully validated by standard Python type checkers.

3.5.1 Detecting Type Conflict. Erroneous type annotations can propagate through dependency chains, leading to a cascade of incorrect inferences for downstream entities. To mitigate this risk, PyTIR employs the static type checker Mypy [25] to identify the erroneous type annotations. The validation process operates on a working copy of the original unannotated source code to isolate the effects of new annotations. Specifically, whenever a type annotation is generated for an entity, PyTIR applies it to this copy. It then invokes Mypy to perform a static analysis on the modified codebase. If Mypy reports any type errors, PyTIR flags the newly added annotation as the source of a *type conflict*, and the annotation is immediately reverted from the working copy.

This incremental and cautious approach prevents faulty annotations from corrupting the type environment, ensuring that each validation step occurs in a consistent context.

3.5.2 Correcting Error Types. Upon detecting a type conflict, PyTIR employs a backtracking strategy to restore type consistency. The procedure retracts the type annotation that caused the conflict and triggers a re-inference process to find a compatible alternative. For most conflicts, PyTIR considers the most recently inferred type to be the source of the conflict. Consequently, this new type is discarded, allowing PyTIR to derive a more suitable type. However, in conflicts between the type of and corresponding parameter in the function definition and between the type of an attribute in a superclass and subclass, the error can originate from either side of the conflict, compelling PyTIR to apply a more sophisticated resolution strategy.

Function parameters present a distinct challenge, as their correct type often depends on usage patterns across the entire codebase. PyTIR handles two primary scenarios. If an overly permissive parameter type leads to invalid operations within the function body (e.g., calling `append()` on a variable typed as `str`), PyTIR provides the specific Mypy error diagnostic as feedback to the LLM, prompting it to perform iterative type narrowing. Conversely, if an overly restrictive parameter type causes type errors at its call sites, the annotation is deemed fundamentally flawed. This triggers an invalidation of the function's annotation. The function is then re-queued for inference, now augmented with the specific conflict context to guide a more accurate, corrective regeneration.

Inheritance hierarchies introduce another layer of complexity, particularly concerning method signature mismatches that violate the Liskov Substitution Principle [23]. For instance, a child method might have a parameter type that is not a valid contravariant of the parent's. To resolve such discrepancies, PyTIR invalidates the annotation of the parent method. The parent method is then submitted for re-inference, with the child's method signature provided as a crucial constraint. This ensures the newly generated parent annotation is compatible, allowing corrections to propagate up the inheritance chain and restore subtyping consistency.

For all other categories of type errors, PyTIR employs a direct feedback loop. The raw error messages from Mypy are fed back to the LLM, which is prompted to generate a refined annotation. This iterative cycle of verification and repair continues until the annotation becomes conflict-free or a predefined iteration threshold is exceeded.

If PyTIR fails to find the suitable type, as a final fallback for persistent conflicts, PyTIR adopts a conservative strategy: it replaces the problematic annotation with the `Any` type. While this sacrifices some type precision, it guarantees the elimination of the type conflict, ensuring that the final output is a fully valid and conflict-free set of annotations.

4 Evaluation

To evaluate the effectiveness of PyTIR, we investigate the following research questions (RQs):

- **RQ1 (Type Accuracy)**: How accurately do the inferred type annotations align with ground truth types?
- **RQ2 (Global Consistency)**: To what extent do generated annotations maintain repository-level consistency across dynamic call resolutions and inheritance hierarchies?
- **RQ3 (Ablation Study)**: What is the contribution of each core component to overall system performance?
- **RQ4 (Efficiency)**: What is the time cost for type inference?

The machine for running experiments is equipped with an Intel Xeon Gold 6330 CPU (28 cores, 2.00GHz), Ubuntu 24.04, and Python 3.10.

4.1 Evaluation Setup

Before the evaluation, we introduce the implementation settings of our approach and the construction of our benchmark dataset, as well as the baselines and metrics used in the evaluation.

4.1.1 Implementation. PyTIR is implemented in 5,000 lines of Python code. During EDG initialization, we leverage PyAnalyzer [22], a SOTA static analysis tool, to precisely identify objects used in each statement, according to which we develop the dependency between entities. To address the context window length limitations of LLMs, we set the maximum entity cluster size to 5 based on the entity lengths observed in our experiments. We allow the LLM to attempt inference on each entity up to three times, stopping when it produces a valid type annotation or when the attempt limit is reached. To comprehensively evaluate our approach, in our ablation study, we employ both a generative model (*DeepSeek-V3*) and a reasoning model (*DeepSeek-R1*), which represent distinct architectural paradigms for type inference benchmarking. For other RQs, we utilize the results derived from *DeepSeek-V3* to represent the performance of PyTIR.

Table 1. The statistics of all repositories in benchmarks

Repository	#Files	#Lines	#Tokens	#Vars	#Funcs	#Entities	Domain
pre_commit_hooks 5.0.0	35	1,501	11,672	62	105	163	Code Quality
flake8 7.3.0	33	2,848	18,452	96	210	498	Code Quality
typer 0.16.0	16	2,156	21,653	133	164	441	CLI*
flask 3.1.1	24	5,645	27,134	110	369	654	Web Development
pre_commit 4.2.0	66	3,816	34,709	236	352	580	Code Quality
fastapi 0.116.1	44	3,742	35,485	137	229	767	Web Development
click 8.2.1	16	6,628	39,319	90	526	879	CLI*
urllib3 2.5.0	36	6,750	41,774	199	492	1,118	Network Request
black 25.1.0	25	6,875	54,479	124	393	681	Code Formatting
jinja2 3.1.6	25	9,074	57,431	166	743	1,495	Web Development
werkzeug 3.1.3	52	13,622	77,033	161	1,113	1,964	Web Development
rich 14.0.0	78	12,819	115,536	372	901	1,980	CLI*

* CLI: Command-Line Interface

4.1.2 Benchmark Construction. To evaluate PyTIR's effectiveness, we construct a high-quality benchmark by selecting 12 projects from TypyBench [21] that meet stringent quality criteria: GitHub stars exceeding 10,000 and type annotation coverage surpassing 85%. To maintain the temporal validity of evaluation, we clone the repositories of these projects, and checkout to their latest release version on PyPI [18]. These repositories preserve developer-authored type annotations, serving as ground truth (R_{truth}) to assess type accuracy. Table 1 summarizes the characteristics of repositories, including size, entities that require type annotation, and domain.

Following TypyBench, we systematically removed all type annotations and type-related docstrings from R_{truth} to produce the unannotated repository $R_{\overline{anno}}$. To accurately evaluate the global type consistency in RQ2, we then performed prerequisite error suppression on $R_{\overline{anno}}$: we ran the static type checker Mypy to detect inherent type errors and, following PEP 484, added '`#type: ignore`' comments to the erroneous statements. This yields an unannotated and error-free (i.e., globally consistent) initial repository R_{raw}, which we use to compare the number of type errors introduced in R_{out} after inferring types with PyTIR. Notably, we configure Mypy to exclude `var-annotated`, `assignment`, and `has-type` error categories, as these stem from incomplete annotations rather than semantic defects, and suppress `missing-imports` errors since they are attributable to third-party dependencies.

4.1.3 Evaluation Metrics. Following TypyBench's methodology, we evaluate PyTIR's performance through two complementary dimensions: type accuracy and global consistency.

Type Accuracy Assessment. We evaluate the accuracy of the type annotations produced by PyTIR by computing two metrics, *TypeSim* and *TypeExact*, between the type-annotated repository R_{out} generated by PyTIR and the ground-truth repository R_{truth} .

- *TypeSim* measures functional similarity between inferred and ground truth types (i.e., types in R_{out} and R_{truth}) based on their shared methods and operations. The similarity score of predicted type T_p and ground truth type T_g is computed as:

$$\text{sim}(T_p, T_g) = \frac{|attrs(T_p) \cap attrs(T_g)|}{|attrs(T_p) \cup attrs(T_g)|}$$

  where $attrs(T)$ denotes the set of methods and operations supported by type T, excluding those common to all types that inherit from `object`. This structural approach prioritizes behavioral equivalence over nominal matching, accommodating Python's duck typing paradigm.
- *TypeExact* provides strict nominal accuracy, requiring exact type identity matches. This serves as a conservative baseline for traditional typing system.

Global Consistency Measurement. PyTIR processes R_{raw} and yields an annotated repository R_{out}. We quantify the degradation in global consistency by executing Mypy on R_{out} and recording the number of reported errors. This metric directly reflects practical barriers developers would encounter when integrating inferred types. For example, a function returning `List[int]` but consumed as `List[str]` would disrupt static type checking, undermining core benefits of type annotations. Such inconsistencies manifest tangible maintenance and refactoring obstacles.

4.1.4 Baselines and Ablation Methods. To the best of our knowledge, PyTIR is the first approach targeting *repository-level* type inference. Consequently, we selected four SOTA *function-level* type inference methods for comparative analysis. Specifically, for RQ1 and RQ2, we employed two *SA-based* methods and two *learning-based* methods as baselines.

HiTyper[30] is a hybrid type inference approach for Python that integrates static analysis with deep learning. It constructs Type Dependency Graphs to model interdependencies among variables within a function body, enabling iterative type propagation.

QuAC[41] employs an attribute-centric methodology. It extracts attribute sets (e.g., accessed methods/properties) from expressions and infers types via information retrieval techniques. This is achieved by matching observed attributes against type-specific attribute sets using BM25 ranking.

TypeGen[31] is a generative type inference framework that combines static analysis with few-shot learning. It generates Chain-of-Thought prompts from code slices and then employs in-context learning with LLMs to predict types.

TIGER[36] introduces a generating-then-ranking framework. It fine-tunes encoder-decoder models with contrastive learning, a generative model produces candidates, and a similarity model ranks them against user-defined types.

For RQ3, we conduct a comprehensive evaluation to assess the impact of PyTIR's two core strategies: EDG-based dependency extraction and backtracking-based type conflict resolution, as well as the integration of LLM. To quantify their contributions, we designed four ablation variants.

$PyTIR^{Base}$ where all strategies are removed. Entity definitions are directly fed to the LLM without dependency information or conflict resolution.

$PyTIR^{W/oDep}$ removes EDG-based dependency extraction. Provides only entity definitions to the LLM while retaining conflict resolution.

$PyTIR^{W/oFix}$ removes backtracking-based conflict resolution. Provides full dependency but skips type conflict correction.

$PyTIR^{R1}$ replaces the LLM used in PyTIR, DeepSeek-V3, with a reasoning model DeepSeek-R1.

4.2 RQ1. Type Accuracy

To conduct a granular analysis of PyTIR's accuracy across different types, we categorize the types used in benchmarks into five distinct groups: basic types (`int`, `float`, `str`, `bytes`, `None`), container types, Union types, user-defined types, and other complex types. The type prediction accuracy of PyTIR and baseline methods across these categories is shown in Fig. 4. The results demonstrate the superior performance of PyTIR. In the overall evaluation, PyTIR achieves a *TypeSim* score of 0.89, representing a significant improvement of 27.1% over the best-performing baseline, TIGER (0.70). Moreover, PyTIR's exact match accuracy alone is 0.84, a score that surpasses all other methods. To better isolate the contribution of our proposed components, we consider the native LLM as a degenerative variant of our approach, $PyTIR^{Base}$, and thus present this comparison between PyTIR and the native LLM in the ablation study in Sec. 4.4.

The *SA-based* methods, HiTyper and QuAC, perform reasonably well on Basic types (0.48 and 0.42, respectively), where direct data-flow analysis is effective. However, they struggle significantly with more ambiguous or complex scenarios like Union types and nested Container types, which often require a deeper semantic understanding of developers' intent. In contrast, the *learning-based* methods, TIGER and TypeGen, generally outperform the static analysis tools, indicating that their models have learned to recognize semantic patterns beyond what rule-based analysis can capture. TIGER, as the strongest baseline, shows respectable performance on UserDef (0.65) and Union (0.61) types. However, their one-shot prediction process, lacking a verification loop, still falls significantly short of PyTIR's accuracy.

Answer to RQ1: PyTIR significantly outperforms the SOTA type inference methods in terms of accuracy. It achieves an overall accuracy of 0.89, with an exact match rate of 0.84. Its strength is especially evident in handling complex, user-defined, and composite types, demonstrating the effectiveness of its LLM-based generation coupled with a rigorous validation mechanism.

4.3 RQ2. Global Consistency

The errors introduced by PyTIR and baseline methods are presented in Table 2. The results clearly demonstrate the global consistency of the type annotations produced by PyTIR. In total, PyTIR introduces only 554 type errors across all repositories, a number substantially lower than all baseline methods. Specifically, PyTIR reduces the number of introduced type errors by 79.1%

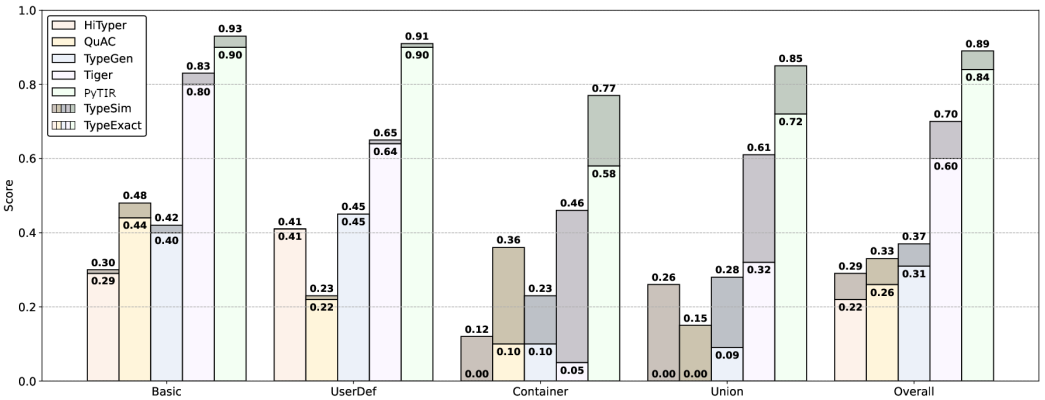


Fig. 4. Accuracy of PyTIR and baseline methods on inference of different categories of types

compared to the best-performing baseline, HiTyper (2,654 errors), and by up to 93.5% compared to TIGER (8,577 errors). This significant reduction underscores PyTIR's effectiveness in generating annotations that are not just locally plausible but also globally coherent. Similar to RQ1, the comparison with the native LLM is presented in our ablation study in Sec. 4.4.

Table 2. Type errors introduced by adding inferred type annotations (Aggregated over 12 repositories). Percentages represent the proportion of each error type relative to the total errors introduced by each method.

Error Type	HiTyper	QuAC	TypeGen	Tiger	PyTIR
arg-type	671 (25.3%)	1,047 (28.1%)	2,923 (34.1%)	1,015 (19.5%)	0 (0.0%)
assignment	403 (15.2%)	849 (22.8%)	1,126 (13.1%)	778 (15.0%)	543 (98.0%)
attr-defined	233 (8.8%)	547 (14.7%)	506 (5.9%)	366 (7.0%)	0 (0.0%)
return-value	325 (12.2%)	385 (10.3%)	616 (7.2%)	190 (3.7%)	0 (0.0%)
call-arg	215 (8.1%)	109 (2.9%)	289 (3.4%)	283 (5.4%)	0 (0.0%)
override	104 (3.9%)	118 (3.2%)	348 (4.1%)	142 (2.7%)	0 (0.0%)
var-annotated	90 (3.4%)	73 (2.0%)	78 (0.9%)	77 (1.5%)	6 (1.1%)
name-defined	7 (0.3%)	74 (2.0%)	1,217 (14.2%)	1,496 (28.8%)	0 (0.0%)
Other Errors[*]	606 (22.8%)	520 (14.0%)	1,474 (17.2%)	855 (16.4%)	5 (0.9%)
Total Errors	**2,654**	**3,722**	**5,202**	**8,577**	**554**

[*] Includes less frequent error types such as '`operator`', '`index`', '`list-item`', '`misc`', etc., detected using Mypy.

A detailed analysis of the error types reveals the inherent challenges of achieving global consistency in automated type inference. A primary challenge is maintaining *inter-procedural consistency*, as a type annotation for a single function must be compatible with every call site across the entire repository. This difficulty is reflected in the high number of arg-type errors observed in baseline methods (ranging from 19.5% to 34.1% of their totals), which often arise when a locally-inferred parameter type conflicts with the type of an argument passed from a different module. In contrast, PyTIR models inter-procedural dependencies using the EDG and employs a type-dependency co-evolution iteration to extract these dependencies. Moreover, for learning-based methods like TypeGen and Tiger, without a mechanism to ensure that a generated type name is valid and accessible in its scope, LLMs may produce plausible types, leading to a large volume of name-defined errors. While PyTIR also leverages an LLM, its validation phase immediately detects and corrects such invalid types. This proactive analysis and reactive validation work in concert, enabling PyTIR to eliminate arg-type, attribute-defined, return-value, and name-defined errors by design.

The errors introduced by PyTIR consist of assignment and var-annotated errors. These two categories of errors are primarily triggered by the absence of type annotations. Consequently, PyTIR omits these errors during iterative refinement to prevent excessive noise in the validation phase. These errors reveal the current limitations of PyTIR's static analysis. However, the introduced errors are few and represent missing or incomplete information rather than the introduction of actively incorrect types. Therefore, the impact of these errors is contained.

Answer to RQ2: PyTIR generates significantly more globally consistent type annotations than SOTA baselines. Its novel validation phase, which iteratively checks and corrects annotations using a live type checker, allows it to effectively eliminate common cross-file and inter-procedural type conflicts that plague other methods. This proactive approach reduces the number of Mypy errors introduced by 79.1% to 93.5%, thereby minimizing the subsequent manual effort required by developers to fix inconsistencies after automatic type annotation.

4.4 RQ3. Ablation Study

As shown in Table 3, the baseline variant, $PyTIR^{Base}$, representing the native LLM, performs the poorest across all metrics. It achieves a TypeExact score of only 0.72 and introduces a substantial 4,186 static errors. This result highlights the inherent limitation of prompting an LLM with isolated code snippets, as it fails to comprehend the intricate web of inter-procedural dependencies, leading to lower accuracy and widespread type inconsistencies.

Table 3. Performance Comparison of PyTIR Variants

	$PyTIR^{Base}$	$PyTIR^{W/oFix}$	$PyTIR^{W/oDep}$	$PyTIR^{R1}$	PyTIR
TypeSim	0.81	0.88	0.87	0.89	0.89
TypeExact	0.72	0.81	0.81	0.83	0.84
Introduced Errors	4,186	3,478	659	335	554

When we introduce only dependency analysis ($PyTIR^{W/oFix}$), type inference accuracy improves substantially. Compared with $PyTIR^{Base}$, TypeSim increases from 0.81 to 0.88 (+0.07), and TypeExact rises from 0.72 to 0.81 (+0.09). However, this variant still introduces 3,478 new type errors, only a 16.9% reduction from 4,186, indicating that dependency context alone cannot guarantee repository-level consistency. In contrast, $PyTIR^{W/oDep}$ introduces conflict resolution, reducing introduced errors from 4,186 to 659 (an 84.3% reduction). Notably, the conflict resolution mechanism also provides dependency information via the backtracking strategy. Therefore, $PyTIR^{W/oDep}$ also improves accuracy over $PyTIR^{Base}$ (TypeSim: 0.81 → 0.87; TypeExact: 0.72 → 0.81). Taken together, these two variants reveal complementary effects: dependency extraction primarily improves prediction quality, whereas conflict resolution is the key driver of global consistency. The full PyTIR, which combines both EDG-based dependency extraction and backtracking-based conflict resolution, achieves the highest performance. It outperforms all variants, reaching a TypeExact score of 0.84 and a TypeSim score of 0.89, and reducing introduced errors to 554.

Finally, we examine the impact of the LLM choice by comparing the full PyTIR (using DeepSeek-V3) with the $PyTIR^{R1}$ variant (using DeepSeek-R1). $PyTIR^{R1}$ achieves a TypeExact score of 0.83, nearly identical to the 0.84 of the standard PyTIR. Moreover, it introduces significantly fewer static errors, reducing 40% from 554 to 335. This substantial improvement in type safety can be attributed to DeepSeek-R1's superior logical reasoning abilities. A more powerful reasoning engine is better at understanding complex semantic relationships and maintaining global consistency, resulting in more reliable and correct type annotations. However, this enhanced reasoning capability comes at a higher computational cost. Therefore, the choice of LLM presents a practical trade-off between inference quality and efficiency. Our default configuration of PyTIR with DeepSeek-V3 represents a balanced choice for general use, while a more powerful reasoning model like DeepSeek-R1 is a viable option for applications.

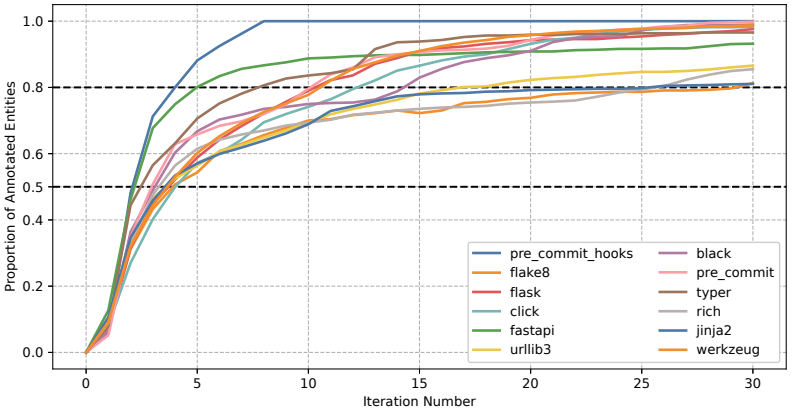


Fig. 5. Annotation rate over iterations across repositories

Answer for RQ3: Both EDG-based dependency extraction and backtracking-based conflict resolution are critical to PyTIR's performance. They have a strong synergistic effect: dependency extraction provides essential context for accurate prediction, while conflict resolution is indispensable for maintaining global type consistency and preventing error propagation. The choice of LLM presents a practical trade-off between inference quality (fewer errors) and computational cost.

4.5 RQ4. Efficiency

To evaluate the execution efficiency of PyTIR, we analyze both the number of iterations and the time cost required for type inference on the benchmark.

As illustrated in Fig. 5, PyTIR demonstrates remarkable efficiency across all 12 repositories. The type annotation process for each repository follows a characteristic pattern: an initial phase of rapid progress followed by a phase of gradual refinement. In the early stages, PyTIR swiftly completes type inference for entities with simple dependencies. The curves in the figure confirm this trend, showing that PyTIR annotates over 50% of entities within the first five iterations for all repositories. Subsequently, the rate of annotation decelerates as the system begins to resolve more complex, long-range dependencies between entities, requiring more feedback cycles.

The total number of iterations required for complete type inference varies with the scale and complexity of each repository. For instance, *pre_commit_hooks*, a smaller repository with only 163 entities, achieves complete annotation in under 10 iterations. In contrast, for larger and more complex repositories such as *urllib3*, *jinja2*, *werkzeug*, and *rich*, which each contain over 1,000 entities, PyTIR requires approximately 30 iterations to surpass 80% annotation coverage. Despite these variations, the process demonstrates robust performance, consistently achieving over 80% annotation coverage for all 12 repositories within 30 iterations.

Moreover, we measured the total execution time for each repository, with detailed results provided in the supplementary material. Across the benchmark, the average time to complete the type annotation process for a single repository is 3.56 hours. Similar to the iteration count, the execution time is correlated with the repository's characteristics. Empirically, we found that the annotation time is positively correlated with both the total number of tokens in the repository, which serves as a proxy for its scale, and the average number of parameters per function, which can indicate its code complexity.

Answer to RQ4: PyTIR's iterative type inference process is highly efficient. It rapidly annotates over 50% of entities within the first five iterations and consistently achieves over 80% coverage within 30 iterations across diverse repositories. The average execution time is 3.56 hours per repository, with the required time and number of iterations being positively correlated with the repository's scale and code complexity. This demonstrates that our approach is practical for real-world applications.

5 Threats to Validity

A primary threat concerns the scalability of PyTIR. The computational cost of PyTIR is correlated to the size and complexity of the target repository. For large and complex repositories, the processing time might become unaffordable. To mitigate this threat, we have designed PyTIR to support an incremental, iterative workflow. Besides processing an entire codebase at once, PyTIR can also be applied on a per-module or per-commit basis during the software development lifecycle.

Another threat lies in the potential presence of pre-existing type errors within real-world repositories. Since PyTIR infers types based on the observed behavior and interactions of entities, type errors in the source code can mislead the inference process, leading to the generation of incorrect type annotations. To address this, during the construction of our evaluation dataset, we meticulously identified and masked potential static type errors. This ensures that our experimental evaluation is conducted on a sanitized baseline, allowing for a fair assessment of PyTIR's inference capabilities without confounding factors from legacy code issues. Moreover, within the PyTIR framework, our backtracking-based conflict resolution mechanism is designed to handle such situations. When an unresolvable type conflict is detected, PyTIR assigns the '`Any`' type, preventing the propagation of an error from one incorrect entity to its dependents. Furthermore, the explicit insertion of '`Any`' serves as a diagnostic signal for developers, flagging regions of the code where underlying type inconsistencies may exist and warrant manual inspection.

6 Related Work

In this section, we provide a comprehensive review of the related work on Python type inference, including *static-analysis-based* and *learning-based* approaches.

6.1 Static-Analysis-Based Type Inference

A line of research in type inference leverages static analysis. These methods operate on a set of pre-defined heuristics and formal rules to deduce variable types [2, 14, 15, 19]. By traversing a program's AST, they propagate type constraints throughout the codebase. This paradigm has given rise to several widely-adopted industrial tools, including Pyright/Pylance [27], Pyre [26], pytype [9], and the official Python type checker, Mypy [25]. Concurrently, numerous academic proposals have explored static type inference not only for Python [29, 42] but also for other dynamically-typed languages like JavaScript [3, 5].

Despite their high precision in predictable code contexts, the efficacy of static analysis-based approaches is often constrained by the inherent dynamism of languages like Python. Features such as dynamic attribute access, metaprogramming, and calls to external, unannotated libraries present significant challenges [30]. Consequently, these methods frequently suffer from low coverage, failing to infer types in a substantial number of real-world scenarios.

6.2 Learning-Based Type Inference

To address the challenges posed by Python's dynamic nature, a growing body of research has shifted towards learning-based methods, which learn to infer types from vast code corpora.

Early efforts in this domain framed type inference as a classification task, training supervised models to predict a type for a given variable from a predefined set of candidates. For instance, TypeWriter [32] combines probabilistic prediction with a search-based refinement process, validating its inferences with a gradual type checker. Lambdanet [39] pioneered the use of graph neural networks (GNNs) on a program's type dependency graph to predict both standard and user-defined types in TypeScript. DLInfer [43] first extracts program slices for variables and then employs a Bi-directional GRU to model type propagation from these contexts. More recently, TypeBert [20] demonstrated that transformer-based models, with their strong inductive bias for token sequences, can achieve state-of-the-art performance when trained on sufficient data. A principal drawback of these classification-based models is their struggle with out-of-vocabulary (OOV) types. This severely limits their practical applicability in real-world projects where new user-defined and library types are prevalent.

The recent success of large pre-trained models for code [6, 7, 11, 37] has inspired a new wave of research. This line of work reformulates type inference as a conditional text generation problem, often as a fill-in-the-blank task. A special placeholder is inserted at the annotation site, and the model is prompted to generate the most probable type based on the surrounding code context. To enhance performance, some studies fine-tune these models on type-annotated codebases [38].

More recently, the advent of LLMs has opened new avenues for type inference. TypeGen [31] exemplifies this trend by utilizing powerful, parameter-frozen LLMs such as GPT-3.5 and GPT-4. It synergizes lightweight static analysis with in-context learning [4] and Chain-of-Thought (CoT) [40] prompting to generate not only type predictions but also their rationales. While demonstrating impressive performance, the reliance on proprietary, resource-intensive LLMs introduces significant practical hurdles, primarily concerning scalability and computational cost, which motivates our work.

7 Conclusion

In this paper, we propose PyTIR, an LLM-powered approach to automatically generate type annotations for Python repositories. PyTIR employs a novel Entity Dependency Graph to model inter-procedural type dependencies in the repository. During each iteration, dependencies and type information co-evolve through mutual refinement. Experimental results show that PyTIR's effectiveness outperforms baseline approaches across all the metrics, achieving higher type inference accuracy with fewer introduced type errors.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (NSFC) under grant No.62132020.

Data Availability

Our replication package is publicly available at https://github.com/SQUARE-RG/PyTIR.

References

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59. doi:10.1007/S10664-022-10278-4

[2] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards type inference for JavaScript. In *European conference on Object-oriented programming*. Springer, 428–452. doi:10.1007/11531142_19

[3] Sheng Chen and Martin Erwig. 2016. Principal type inference for gadts. *ACM SIGPLAN Notices* 51, 1 (2016), 416–428. doi:10.1145/2914770.2837665

[4] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, et al. 2024. A Survey on In-context Learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 1107–1128. doi:10.18653/V1/2024.EMNLP-MAIN.64

[5] Michael Emmi and Constantin Enea. 2016. Symbolic abstract data type inference. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 513–525. doi:10.1145/2914770.2837645

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547. doi:10.18653/V1/2020.FINDINGS-EMNLP.139

[7] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[8] GitHub. 2024. Octoverse 2024 – The state of open source. https://github.blog/news-insights/octoverse/octoverse-2024/, Last accessed on 2025-08-01.

[9] Google. [n. d.]. pytype: A static type analyzer for python code. https://github.com/google/pytype

[10] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 – Type Hints. https://peps.python.org/pep-0484/, Last accessed on 2025-08-01.

[11] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225. doi:10.18653/V1/2022.ACL-LONG.499

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[13] Aric Hagberg and Drew Conway. 2020. Networkx: Network analysis with python. *URL: https://networkx. github. io* (2020), 1–48.

[14] 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. doi:10.1007/978-3-319-96142-2_2

[15] Mostafa Hassan, Caterina Urban, and Peter Muller. 2017. SMT-based static type inference for Python 3. *Bachelor thesis, ETH Zürich, Department of Computer Science* (2017).

[16] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[17] IEEE Spectrum. 2024. The Top Programming Languages 2024. https://spectrum.ieee.org/top-programming-languages-2024, Last accessed on 2025-08-01.

[18] Python Package Index. 2025. PyPI. https://pypi.org/

[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255. doi:10.1007/978-3-642-03237-0_17

[20] Kevin Jesse, Premkumar T Devanbu, and Toufique Ahmed. 2021. Learning type annotation: Is big data enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1483–1486. doi:10.1145/3468264.3473135

[21] Yuhe Jiang, Xun Deng, Jiacheng Yang, Honghua Dong, Gennady Pekhimenko, Fan Long, and Xujie Si. [n. d.]. TypyBench: Evaluating LLM Type Inference for Untyped Python Repositories. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

[22] Wuxia Jin, Shuo Xu, Dawei Chen, Jiajun He, Dinghong Zhong, Ming Fan, Hongxu Chen, Huijia Zhang, and Ting Liu. 2024. Pyanalyzer: An effective and practical approach for dependency extraction from python code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3640325

[23] Barbara H Liskov and Jeannette M Wing. 1994. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 6 (1994), 1811–1841. doi:10.1145/197320.197383

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[25] Vasilii Lupuliak. 2025. Static typing in Python. (2025).

[26] Meta. [n. d.]. Pyre: A performant type-checker for python 3. https://pyre-check.org/.

[27] Microsoft. 2022. Pyright: Static type checker for python. https://microsoft.github.io/pyright

[28] Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 585–589. doi:10.1109/MSR52588.2021.00079

[29] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2021. Data flow refinement type inference. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–31. doi:10.1145/3434300

[30] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering*. 2019–2030. doi:10.1145/3510003.3510038 arXiv:2105.03595 [cs].

[31] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 988–999. doi:10.1109/ASE56229.2023.00031 ISSN: 2643-1572.

[32] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220. doi:10.1145/3368089.3409715

[33] 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*. 57–70. doi:10.1145/3426422.3426981

[34] Armin Ronacher. 2025. Click. https://github.com/pallets/click Accessed: 2025-9-10.

[35] Armin Ronacher. 2025. Flask. https://github.com/pallets/flask Accessed: 2025-9-10.

[36] Chong Wang, Jian Zhang, Yiling Lou, Mingwei Liu, Weisong Sun, Yang Liu, and Xin Peng. 2025. TIGER: A Generating-Then-Ranking Framework for Practical Python Type Inference. 321-333 pages. doi:10.1109/ICSE55347.2025.00019

[37] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. doi:10.18653/V1/2021.EMNLP-MAIN.685

[38] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference using Static Analysis. In *The Eleventh International Conference on Learning Representations*.

[39] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*.

[40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837. doi:10.52202/068431-1800

[41] Jifeng Wu and Caroline Lemieux. 2024. QuAC: Quick Attribute-Centric Type Inference for Python. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 2040–2069. doi:10.1145/3689783

[42] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 607–618. doi:10.1145/2950290.2950343

[43] Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. Dlinfer: Deep learning with static slicing for python type inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2009–2021. doi:10.1109/icse48619.2023.00170

[44] Yi Yang, Ana Milanova, and Martin Hirzel. 2022. Complex Python features in the wild. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 282–293. doi:10.1145/3524842.3528467