Type-aware LLM-based Test Generation for Python Programs

RUNLIN LIU, Beihang University, China
ZHE ZHANG, Beihang University, China
YUNGE HU, Beihang University, China
YUHANG LIN, Beihang University, China
XIANG GAO*, Beihang University, China
HAILONG SUN, Beihang University, China

Automated test generation has been extensively explored, yet generating high-quality tests for Python programs remains particularly challenging. Because of Python's dynamic typing features, existing approaches, ranging from search-based software testing (SBST) to recent LLM-driven techniques, are often prone to type errors. Hence, existing methods often generate invalid inputs and semantically inconsistent test cases, which ultimately undermine their practical effectiveness. To address these limitations, we present TEST4PY, a novel framework that enhances type correctness in automated test generation for Python. TEST4PY leverages the program's call graph to capture richer contextual information about parameters, and introduces a behavior-based type inference mechanism that accurately infers parameter types and constructs valid test inputs. Beyond input construction, TEST4PY integrates an iterative repair procedure that progressively refines generated test cases to improve coverage. In an evaluation on 183 real-world Python modules, TEST4PY achieved an average line coverage of 83.0% and branch coverage of 70.8%, outperforming state-of-the-art tools by 7.2% and 8.4%, respectively.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: Automated test generation, large language models, type inference, retrieval-augmented generation, test case repair, software testing

ACM Reference Format:
Runlin Liu, Zhe Zhang, Yunge Hu, Yuhang Lin, Xiang Gao, and Hailong Sun. 2025. Type-aware LLM-based Test Generation for Python Programs. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

High-quality testing is crucial in modern software engineering, it not only validates functional correctness but also provides long-term regression safeguards as software evolves. However, manually constructing such tests remains labor-intensive and error-prone [53], motivating extensive research on automated test generation. Classical approaches include random-based techniques [9, 45],

*Corresponding Author.

Authors' Contact Information: Runlin Liu, Beihang University, Beijing, China, runlin22@buaa.edu.cn; Zhe Zhang, Beihang University, Beijing, China, zhangzhe2023@buaa.edu.cn; Yunge Hu, Beihang University, Beijing, China, hygchn04@gmail.com; Yuhang Lin, Beihang University, Beijing, China, yuhanglin35@gmail.com; Xiang Gao, Beihang University, Beijing, China, xiang_gao@buaa.edu.cn; Hailong Sun, Beihang University, Beijing, China, sunhl@buaa.edu.cn.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

constraint-driven strategies [15, 18, 24], and search-based software testing (SBST) [2, 19, 22, 57]. More recently, advances in large language models (LLMs) have further energized this line of work. For example, CodaMosa [19] augments SBST with LLM-generated tests, while CoverUp [1] leverages coverage-guided prompting to iteratively steer LLMs toward higher-coverage test suites.

Despite these encouraging developments, automated test generation remains particularly challenging for dynamically typed languages such as Python. In contrast to statically typed languages, Python omits mandatory type annotations, which obscures parameter types and class affiliations. Empirical studies underscore the severity of this limitation: nearly 30% of developer-reported issues on GitHub and Stack Overflow arise from type-related errors [31]. Since the validity and semantic soundness of inputs are highly important in test cases, type correctness is indispensable for generating meaningful and effective test cases. As discussed in prior work [22], consider the function $triangle(a, b, c)$ intended to determine whether three sides form an equilateral triangle. While the parameters are expected to be **int** or **float**, Python's permissive typing allows structurally valid but semantically irrelevant inputs, such as lists of strings. Such cases may execute without failure and even attain full code coverage, yet they are semantically meaningless, provide no practical regression protection, and ultimately undermine the utility of the generated tests.

Unfortunately, existing approaches exhibit limited accuracy in generating tests with correct types. This deficiency arises from two key factors: (i) parameter construction often lies outside function bodies, depriving models of essential local context; and (ii) user-defined types are highly project-specific and sparsely represented in pretraining corpora, severely restricting model generalization. These limitations crystallize into a fundamental research gap: the lack of mechanisms to generate semantically valid test inputs for dynamically typed languages.

This gap motivates the following research questions:

- How can parameter types be accurately inferred for Python functions in the absence of explicit type annotations and under the constraints of dynamic typing?
- How can program context be effectively constructed and exploited to guide LLMs in synthesizing valid and semantically meaningful test inputs, particularly for complex user-defined types?

Our approach. To address these challenges, we present TEST4PY, a framework that infers function parameter types prior to test case generation, thereby enhancing the type correctness of the generated test inputs. Although Python's dynamic typing provides considerable flexibility, it also introduces a fundamental obstacle to precise type inference. To overcome this limitation, TEST4PY draws inspiration from the classic "Duck Test"[1], and introduces behavior-guided parameter inference (BGPI). BGPI conceptualizes type resolution as the process of identifying a parameter's type through its observable behaviors at both the syntactic and semantic levels. However, parameters in real-world software rarely exist in isolation; they interact through intricate call relationships spanning multiple modules. To model such dependencies, we construct a project-level call graph and derive parameter-centric semantic summaries from both callee behaviors and caller contexts, thereby enabling context-aware semantic inference.

Subsequently, TEST4PY leverages the inferred types to construct a compact but expressive *type context*. This enriched representation guides LLMs in synthesizing executable and semantically valid test inputs. To further improve robustness, TEST4PY introduces an adaptive error repair mechanism that autonomously performs error resolution.

We evaluate TEST4PY on 183 real-world Python modules, and compare it with CodaMosa [19] and CoverUp [1]. Evaluation results show that TEST4PY achieves an average line coverage of 83.0% and branch coverage of 70.8%, surpassing state-of-the-art baselines by 7.2% and 8.4%, respectively.

[1] https://en.wikipedia.org/wiki/Duck_test

An ablation study further reveals that the proposed type inference component alone improves coverage by 13.5% in settings lacking explicit type annotations.

In summary, this paper makes the following contributions:

- We propose a novel approach for parameter-centric summarization that leverages a combination of callee-driven behaviors and caller-contextual evidence to enhance the type-correctness and semantic-correctness of generated test cases.
- We propose a behavior-guided type inference framework that integrates syntactic filtering with semantic retrieval to enhance inference accuracy, and further exploits the inferred type information to enhance the quality of LLM-based Python test case generation. We have released the source code of TEST4PY on https://doi.org/10.5281/zenodo.18719652.
- We conduct a comprehensive evaluation on real-world projects, demonstrating significant improvements in line and branch coverage over state-of-the-art baselines.

2 Motivation

We use the `get_custom_loader` function from the `PyCG` [2] project as a representative example to illustrate our motivation. `PyCG` is an open-source project on GitHub with hundreds of stars. It generates call graphs by analyzing Python code and supports advanced features such as higher-order functions and complex class inheritance structures.

```python
def get_custom_loader(ig_obj):
  class CustomLoader(importlib.abc.
      SourceLoader):
    def __init__(self, fullname, path):
      self.fullname = fullname
      self.path = path
      ig_obj.create_edge(self.fullname)
      if not ig_obj.get_node(self.
          fullname):
        ig_obj.create_node(self.fullname)
        ig_obj.set_filepath(self.fullname
            , self.path)
    <...omitted code...>
  return CustomLoader
```

a: Part of the get_custom_loader function

```python
def test_custom_loader():
  ig_obj=MockImportGraph()
  loader=get_custom_loader(ig_obj)
  <...omitted code...>
```

b: Test case Generated by LLM

```python
def test_case_4():
  try:
    bool_0 = True
    var_0 = module_0.
        get_custom_loader(bool_0)
    <...omitted code...>
  except BaseException:
    pass
```

c: Test case Generated by SBST

```python
class ImportManager(object):
  def create_node(self, name):
    self.import_graph[name] = {
    "filename":"", "imports":set()}

  def set_filepath(self, node_name,
      filename):
    node = self.get_node(node_name)
    node["filename"] = os.path.
        abspath(filename)
```

d: Part of the ImportManager class

Listing 1. An example of code and the corresponding auto-generated unit tests.

As shown in Listing 1a, the function `get_custom_loader` returns a module loader `CustomLoader`. During initialization, the parameter `ig_obj`, which is an instance of the `ImportManager` class presented in Listing 1d, is responsible for maintaining the relationships among imported modules. Specifically, it first establishes the import edges and subsequently verifies whether the target module

[2] PyCG: https://github.com/vitsalis/PyCG

already exists in the import graph. If the module is not present, a new node is created, and the corresponding file path is recorded.

We employ the unit test generation tool CodaMosa [19] to generate test cases for the function `get_custom_loader`. Initially, CodaMosa utilizes search-based software testing (SBST) approach to produce simple test cases, as shown in Listing 1c. This test case sets `bool_0` to `True` and passes it to `get_custom_loader`. However, since the constructor of `CustomLoader` accesses members of `ig_obj`, when `ig_obj` is initialized as boolean, the execution of `CustomLoader`'s instantiation will result in a runtime error, preventing further improvements in coverage. When SBST fails to enhance coverage, CodaMosa activates the large language model to generate test cases. One of the test cases generated by LLM is shown in Listing 1b. In this case, LLM mocks a class named `MockImportGraph` and uses its instance as the parameter of `get_custom_loader`. `MockImportGraph` class ensures that all members accessed by the `__init__` function of `CustomLoader` are present, thus preventing run-time errors. However, due to the absence of explicit type information for `ig_obj`, LLM is unable to infer the exact implementations of methods such as `create_node`. Instead, it relies solely on method names to approximate their behavior, leading to inconsistencies between the generated test case and the actual implementation. CodaMosa then makes 16 additional attempts, all of which fail. The primary cause of this failure is that CodaMosa does not provide type information in its prompts, preventing LLM from generating semantically valid test cases.

To infer the type of `ig_obj`, we employ HiTyper [34], which is the state-of-the-art tool for type inference. However, HiTyper incorrectly infers `ig_obj` as a `str` type. This incorrect inference arises because `ig_obj` is a function parameter without any direct assignment statements to aid type inference. Moreover, since `ig_obj` is a user-defined type, the deep learning component of HiTyper struggles to infer its type accurately.

```
class TestGetCustomLoader(unittest.TestCase):
  def setUp(self):
    self.ig_obj = ImportManager()
    self.loader_class = get_custom_loader(self.
        ig_obj)

  def test_loader(self):
    loader = self.loader_class("test.module",
      "/path/to/module.py")
    self.assertIn("test.module", self.ig_obj.
        import_graph)
    <...omitted code...>
```

Listing 2. Test case generated by Test4Py

In contrast, our proposed Test4Py employs a type inference mechanism to determine the type of the argument `ig_obj`. In the `__init__` function of `CustomLoader`, `ig_obj` invokes member methods such as `create_edge` and `get_node`, which are responsible for maintaining module import relationships. To resolve the type of `ig_obj`, Test4Py identifies classes that define these methods and are semantically related to import management. It subsequently locates the `ImportManager` class and extracts information about it. By incorporating this information into the prompt, LLM can instantiate an `ImportManager` object and pass it as an argument to `get_custom_loader`, as demonstrated in Listing 2. This approach enables Test4Py to successfully instantiate `loader_class` and achieve full line coverage for `get_custom_loader`, thereby not only improving test coverage but also generating assertions that verify the core functionality of the function.

This example demonstrates that accurately inferring type information for key variables can significantly enhance both the coverage of generated test cases and the quality of assertions. Existing approaches, when handling dynamically typed languages like Python, frequently overlook critical contextual information, resulting in poor-quality test cases generated by LLM. Addressing this limitation requires a method capable of searching across the entire project, filtering relevant

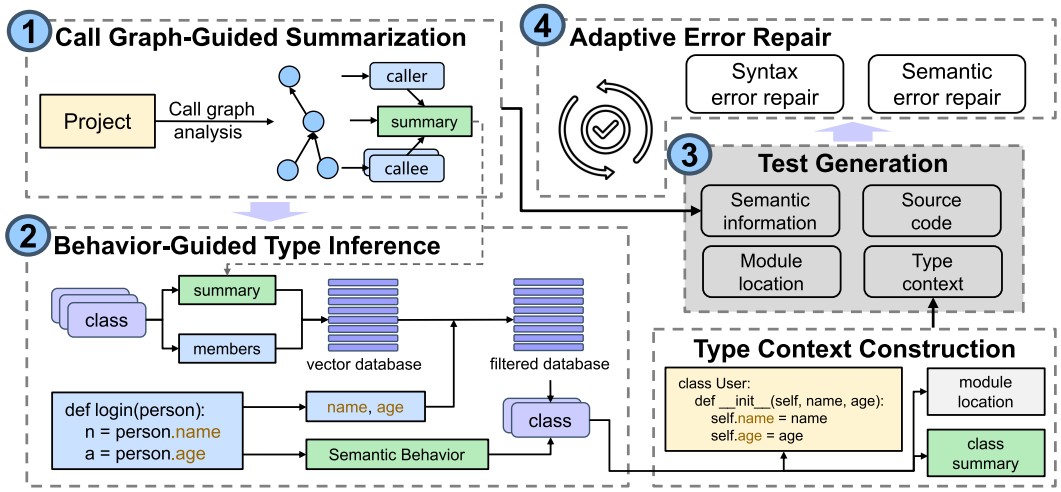

Fig. 2. The overall architecture of Test4Py, comprising four interdependent stages: call graph-guided parameter-centric summarization, behavior-guided parameter type inference, type-guided test case generation, adaptive error repair.

information, and accurately inferring variable types. Test4Py fulfills this requirement, making it a promising solution for improving automated test generation in dynamically typed languages.

3 Methodology

This section presents Test4Py, a framework designed to advance automated unit test generation for Python programs by systematically addressing the challenges posed by dynamic typing and limited type annotations.

3.1 Framework Design

As illustrated in Figure 2, the workflow of Test4Py is structured into four tightly coupled stages, each addressing a fundamental obstacle in unit test generation for dynamically typed languages, where explicit type annotations are scarce and semantic relations among program entities are often implicit.

First of all, to enhance the semantic comprehension of function parameters, Test4Py constructs a project-wide call graph at stage ①. This structural representation enables the derivation of parameter-centric semantic summaries, capturing how each function interacts with its surrounding context. These summaries provide enriched contextual knowledge that significantly improves the accuracy of subsequent inference steps. Subsequently, building upon the function-level summaries, at stage ②, Test4Py derives class-level semantic summaries by aggregating the behaviors of their constituent methods. These class summaries are then embedded and stored in a vector database, forming the foundation for whole-project type inference. For each function parameter, syntactic behavior constraints are applied to eliminate incompatible candidates, after which semantic behavior retrieval aligns the observed parameter usage with the most plausible classes. This explicit coupling of structural and semantic evidence goes beyond conventional inference approaches, yielding type predictions that are both precise and contextually grounded. Then, once candidate classes are inferred, at stage ③, Test4Py constructs a comprehensive type context that integrates the class constructor, module path, and functional summary. This enriched context serves as the foundation

for LLM-based test generation, guiding the model to synthesize more faithful and executable test cases. To further enhance robustness, stage ④ incorporates an adaptive error-repair mechanism that automatically detects and corrects both syntactic flaws and semantic inconsistencies in the generated tests.

3.2 Call Graph-Guided Parameter-Centric Summarization

Motivation. Recent studies have shown that LLM-generated natural language summaries of functions can enhance automated test case generation [55], as they provide richer semantic representations of the functions under test. However, existing function summarization approaches are not designed for test generation task. For test generation, the most critical semantic unit is the function parameter. Parameters embody both the interface contract and the input-output behaviors of software components, and inaccuracies in parameter understanding directly compromise the validity of generated tests. We therefore advocate a parameter-centric perspective: our goal is to produce summaries that explicitly model parameter types, constraints, and usage intents. Yet, parameters in real-world functions are rarely self-contained; their semantics emerge through complex inter-procedural interactions, where callees constrain permissible parameter values and callers project realistic usage patterns. Without capturing these dependencies, LLM-generated summaries risk being incomplete or misleading.

Approach. To address this challenge, TEST4PY introduces a *Call Graph-Guided Parameter-Centric Summarization* framework, where project-level call graph analysis serves as the structural backbone to integrate two complementary sources of parameter semantics: (i) *callee-driven behavioral constraints*, which reveal how parameter values influence downstream computations; and (ii) *caller-contextual evidence*, which illuminates the concrete types and intents associated with parameter usage. We adopt `pycg` [37], a state-of-the-art static call graph construction framework, to ensure reliable extraction of inter-procedural dependencies.

3.2.1 Callee-Driven Behavioral Analysis. This phase focuses on how parameters propagate effects across the call chain. The summary of a callee not only provides the behavioral semantics represented by the call expressions but also reveals the relational semantics among its parameters. For example, if a callee checks whether two arguments are equal, this constraint informs that the parameters may satisfy an equality condition. Capturing such parameter-centric constraints requires analyzing not only immediate callees but also deeper invocations along the call chain. A simple traversal of direct function calls is therefore insufficient, as it would omit semantic information carried through deeper dependencies. To overcome this limitation, we adopt a topological sorting approach, where each function's summarization incorporates the summaries of the functions it calls. This method allows us to capture and refine parameter-related semantics across the entire call chain.

Formally, let $Called(f)$ denote the set of functions invoked by f, and $f.sc$ its source code. Given an instruction $be\_instruction$, which directs LLM to analyze the roles of function parameters and how modifications to their values influence the execution of the function, the forward topological summary of f is defined as:

$$f.be\_summary = \text{LLM}\left(be\_instruction + f.sc + \sum_{x \in Called(f)} x.be\_summary\right). \tag{1}$$

To handle recursion, cycles in the call graph are temporarily relaxed by removing one dependency edge, ensuring termination while preserving semantic integrity.

3.2.2 Caller-Contextual Semantic Projection. Although forward traversal effectively captures the fine-grained semantics of callees, it inherently overlooks the *usage-driven* semantics that arise from domain-specific conventions, how developers in a particular application domain consistently interpret and utilize functions within conventional usage contexts. For example, a function invocation *triangle(3, 4, 5)* reflects the semantics of determining whether three integers compose a triangle, implying that its parameters represent numeric types such as `int` or `float`. To address this, we introduce a *caller-contextual summarization* strategy that traverses the call graph in reverse topological order, treating each function as a semantic construct shaped by the behavioral constraints imposed by its callers and their domain-specific usage conventions.

This strategy fulfills two complementary objectives:

- **Parameter Type Inference.** Caller-contextual evidence enables more accurate preliminary parameter type inference by leveraging concrete invocation patterns. For example, if a caller invokes a function as `add(1, 2)`, LLM can infer that both parameters are likely of type `int`.
- **Semantic Intent Modeling.** Beyond type inference, caller-contextual projection facilitates the inference of semantic intent. This helps LLM in generating test cases that are better aligned with realistic usage scenarios and domain-specific conventions.

Moreover, for functions with zero in-degree in the call graph (i.e., top-level entry points), TEST4PY augments the summarization process with auxiliary resources such as project-level documentation (e.g., `README.md`). This supplementation establishes domain-level semantics that cannot be derived solely from intra-project call relationships.

Formally, let $Call(f)$ denote the set of functions invoking f, and *docs* denote the project-level documentation. Given an instruction *se_instruction*, which directs LLM to analyze the semantic roles and types of function parameters, the reverse summary is defined as:

$$f.se\_summary = \begin{cases} \text{LLM}(se\_instruction + x.sc + x.se\_summary), & Call(f) \neq \emptyset,\ x \in Call(f) \\ \text{LLM}(se\_instruction + docs), & Call(f) = \emptyset. \end{cases} \tag{2}$$

Due to the limited context window of LLMs, it is impractical to include all elements of $Call(f)$ in the prompt. Consequently, certain semantic cues from unobserved callers may be omitted, leading to an incomplete *se_summary*. To address this limitation, the generated *se_summary* is treated as an auxiliary semantic reference, rather than as a strict constraint on semantics in subsequent stages.

3.2.3 Summary Synthesis. Finally, TEST4PY synthesizes $f.be\_summary$ and $f.se\_summary$ to produce the final summary $f.summary$. This synthesis step removes redundant information and aligns fine-grained behavioral details with higher-level abstractions, resulting in a concise yet comprehensive semantic representation that mitigates redundancy and prevents information overload in subsequent steps.

Example. To demonstrate how our approach integrates behavioral and semantic cues for precise parameter understanding, we present a representative example. Figure 3a depicts the function `transform` along with its caller and callee. The function aims to scale all `Record` objects contained in `x` by a factor of y.

The *callee-driven behavioral analysis* exploits the summary of `process_all`, which encodes its iteration and scaling behaviors over `Record` objects. From this behavioral evidence, the model infers that the parameter x must support both `scale` and `average` methods, as captured in the *be_summary* (Figure 3b). Subsequently, the *caller-contextual semantic projection* analyzes the invocation context of `transform` in `test`, where the concrete arguments reveal that x is of type `List[Record]`, as shown in the *se_summary* (Figure 3c).

```python
def transform(x, y):
    results = process_all(x, y)
    <...omitted code...>

def process_all(objs, scale_factor):
    for o in objs:
        o.scale(scale_factor)
    return [o.average() for o in objs]

def test():
    r1 = Record([1, 2, 3])
    r2 = Record([10, 20])
    transform([r1, r2], 2.5)
```

a: The function `transform` and its related functions.

```
x (list): A list of objects to be
    transformed. Each object must have
    `scale` and `average` methods.
y (float): The scaling factor used to
    transform the objects.
```

b: Behavioral summary (*be_summary*) of `transform`.

```
x (List[Record]): A list of Record
    objects.
y (float): A scaling factor used to
    scale each Record object in the
    list.
```

c: Semantic summary (*se_summary*) of `transform`.

Fig. 3. Example demonstrating how TEST4PY synthesizes behavioral and semantic summaries.

However, due to the dynamic language features inherent to Python, no single static analysis algorithm can generate a perfect call flow graph. Even the robust pycg framework exhibits a recall rate of only 69.9%. This fundamental limitation indicates that relying solely on the Call Graph-Guided summarization method is insufficient to fully resolve the parameter type inference problem in Python programs.

3.3 Behavior-Guided Parameter Type Inference

As highlighted in Section 1, accurately resolving the types of dynamically defined function parameters remains a fundamental obstacle to automated test case generation. The challenge is particularly acute for user-defined types with intricate internal structures, where traditional inference strategies fail to capture the semantic dependencies between program entities and their contextual usage. Existing methods either depend on incomplete annotations or rely on shallow syntactic heuristics, both of which exhibit limited effectiveness in dynamically typed, real-world software systems.

To address this limitation, we introduce the principle of *behavior-guided inference*. The central hypothesis is that the observable operational behaviors of a parameter—such as the members it accesses and the operations it supports—provide the most discriminative evidence for type identification. This perspective resonates with the intuition underlying the classical "Duck Test"[3], which informally asserts that an entity can be identified by its characteristic behaviors. We reformulate type inference as an alignment problem between behavioral evidence and candidate type definitions, thereby extending the heuristic notion into a principled retrieval-augmented framework. Building on this insight, we propose the *Behavior-Guided Parameter Inference (BGPI)* framework, which integrates syntactic behavior filtering with semantic behavior retrieval to achieve precise and context-aware type resolution.

The overall workflow is summarized in Algorithm 1. Parameters are categorized into three groups: (1) built-in Python types (e.g., `int`, `List`), (2) third-party library types (e.g., `ast.FunctionDef`), and (3) project-specific user-defined types. The first two categories can often be inferred by LLMs through prior knowledge acquired from pretraining. The central challenge lies in resolving user-defined types, which are inherently underrepresented in training corpora and thus require project-specific reasoning.

TEST4PY first builds a knowledge base of all project-specific classes. Specifically, it applies static analysis (Lines 4-6) to extract all accessible member variables and methods for each class. To

[3]https://en.wikipedia.org/wiki/Duck_test

Algorithm 1: Behavior-Guided Type Resolution for Parameter

Input: Target program *program*
Output: Type for all parameters

```
1  vector_database ← ∅;
2  result ← ∅;
3  foreach class ∈ program do
4      class.members ← GetMembers(class);
5      foreach super ∈ superClass(class) do
6          class.members ← class.members ∪ GetMembers(super);
7      class.summary ← Summarize(class);
8      vector_database ← vector_database ∪ Vectorize(class.summary);
9  foreach f ∈ program do
10     foreach para ∈ f.paras do
11         if HasTypeAnnotation(para) then
12             type ← GetTypeAnnotation(para);
13         else
14             type ← InferType(f, para);  // Preliminary inference
15             if type is user-defined then
16                 members ← FindMembers(para, f);  // Members accessed in f
17                 filtered ← FilterByMembers(vector_database, members);
18                 type ← SemanticRetrieve(f, para, filtered);
19         result ← result ∪ {(para, type)};
20 return result
```

mitigate the limitations of raw code embeddings, each class is further abstracted by an LLM into a concise functional summary (Line 7), which captures its high-level semantics while filtering out syntactic noise. These summaries are encoded into semantic vector representations (Line 8) and stored in a vector database, forming a knowledge base of user-defined types.

At the inference stage, Test4Py performs an initial type inference for each parameter lacking explicit annotations (Line 14). Test4Py performs behavior-guided parameter inference *BGPI* by leveraging the knowledge base gathered in the preceding steps. First, it analyzes the function body to identify all members accessed by each parameter, including field accesses and method invocations (Line 16), and uses these signals to filter candidate classes in the vector database by enforcing syntactic compatibility (Line 17). Subsequently, the refined candidate set is ranked via semantic retrieval, aligning the parameter's observed usage behaviors with the most plausible user-defined type (Line 18). These two stages, which we later formalize as *Syntactic Behavior Filtering* and *Semantic Behavior Retrieval*, jointly ensure both precision and generality in type resolution.

3.3.1 Syntactic Behavior Filtering. The *syntactic behavior filtering* exploits fine-grained structural cues observable in the function body to filter out incompatible types. Specifically, the parameter usage behaviors, which manifest as (1) field accesses and (2) method invocations, serve as discriminative signals that enable early pruning of infeasible type candidates. Formally, given a parameter *para* in function f and a type *class*, let $AccessMembers(para)$ denote the set of accessed members and $DefineMembers(class)$ denote the members defined or inherited by *class*. A necessary

condition is:

$$\forall x \in AccessMembers(para),\ (x \notin DefineMembers(class) \implies typeof(para) \neq class) \quad (3)$$

This constraint enables early elimination of incompatible candidate types.

We implement this mechanism through Abstract Syntax Tree (AST) analysis. For each class, we recursively collect all explicitly defined and inherited members to construct $DefineMembers(class)$. For each parameter, we parse the function body to extract its accessed members to construct $AccessMembers(para)$. In addition, we incorporate specialized handling for Python's built-in object members (e.g., *__dict__*, *__class__*) to prevent spurious eliminations.

3.3.2 Semantic Behavior Retrieval. While syntactic filtering effectively reduces the search space, it alone is insufficient for disambiguating user-defined types. To achieve finer-grained discrimination, we introduce *semantic behavior retrieval*, which leverages the reasoning capabilities of LLMs to infer the semantic role of a parameter from its functional usage context. For example, in a call `login(student)`, the parameter `student` is more plausibly associated with a domain-specific entity (e.g., a student object) rather than a primitive type. To operationalize this idea, the knowledge base of user-defined types is constructed and indexed through *summary-based semantic representations*. Unlike raw code embeddings, which often contain syntactic noise and hinder retrieval accuracy [11], these high-level summaries serve as *semantic anchors* that abstract away irrelevant implementation details while retaining discriminative contextual information. The summaries are generated with the assistance of pre-computed function-level descriptions ($f.summary$), ensuring that the semantic representation of a class faithfully reflects the behavioral cues embodied in its constituent methods. During inference, TEST4PY prompts LLM to synthesize a semantic query that encapsulates the behavioral role of the target parameter *para* within the enclosing function. The query is executed against the filtered vector database using Maximal Marginal Relevance (MMR) [26], which balances relevance and diversity to produce a ranked list of candidate classes. This design not only improves retrieval precision but also mitigates the risk of overfitting to a narrow subset of types.

Moreover, another critical challenge arises from the non-uniqueness of parameter types in Python. Polymorphism allows multiple subclasses of a parent class to be valid for the same parameter, while Python's dynamic typing system further admits structurally or semantically heterogeneous alternatives. Ignoring semantic variability would constrain test generation, thereby limiting execution path exploration and reducing achievable code coverage. To address this problem, TEST4PY retains the top-k retrieved classes based on similarity scores, where k is a configurable parameter. This relaxation strategically broadens the candidate type space, enhancing the robustness and diversity of generated test cases. Overall, semantic behavior retrieval constitutes a pivotal component of our framework, substantially improving the realism of inferred types and strengthening the exploratory capacity of subsequent test generation.

3.4 Type-Guided Test Case Generation

While our framework is able to infer parameter types for the function under test, directly using these inferred types as prompts is insufficient to ensure high-quality test generation. On the other hand, incorporating the complete source code of all potential parameter-related classes would substantially inflate the prompt length and impair the effectiveness of the attention mechanism, thereby reducing the reliability of the generated outputs. This section introduces a concise yet effective approach to leveraging parameter type information, which reduces spurious generations while preserving semantic fidelity.

3.4.1 Prompt Design for Test Generation. To achieve a concise yet informative prompt design, we structure the type context around three complementary components, each targeting a critical aspect of parameter utilization in test case generation:

- **Module Location.** Specifies the module path of the class, ensuring correct dependency resolution through accurate imports.
- **Constructor Signature.** Captures the instantiation logic of the parameter, providing essential information for creating valid objects.
- **Functional Summary.** Encodes the expected behavioral semantics of the parameter within the function under test, guiding the generation of semantically consistent interactions.

This design encapsulates both structural and behavioral perspectives of parameter usage, thereby enriching the contextual signal available to LLM while maintaining a bounded prompt length. The combination of these three elements enables the synthesis of test cases that are both executable and semantically aligned with the intended functionality.

Nevertheless, the semantic retrieval phase of type inference may introduce errors, and such erroneous type contexts can inject substantial noise into the downstream test case generation process. To mitigate this risk, we introduce an *LLM-as-a-Critic* mechanism that evaluates the relevance of candidate type contexts prior to generation. Instead of inferring parameter types directly, LLM is tasked with validating whether a given type context plausibly corresponds to the parameter at hand. For example, in the function $login(student)$, if the suggested type context corresponds to a non-entity utility class, the critic model can readily detect the inconsistency. This validation step substantially improves the robustness of type utilization, thereby enhancing both inference accuracy and the fidelity of generated test cases.

While the preceding step focuses on parameter-type contexts, we complement it with an overall prompt design that captures broader aspects of test case generation. At the system level, we further specify a role-based prompt that explicitly conditions LLM to act as an expert Python developer. Empirical studies indicate that role specification improves the consistency of generated code [46]. Accordingly, our prompt specifies the target function's module path to guarantee dependency correctness, enforces `pytest`-compliant test generation, and provides explicit step-by-step task instructions to improve the reliability of LLM-based synthesis. Moreover, the prompt integrates the functional summary ($f.summary$) from Section 3.2 and the parameter type context extracted in the preceding step, guiding the model toward generating test cases that are both semantically accurate and executable. The final prompt is illustrated in Figure 4.

Prompt Template for Test Case Generation

You are an AI assistant that generates high -quality pytest test cases for Python functions. Analyze the target function before writing the test cases. Import the function using its provided module name. Ensure the following:

Type Awareness: Generate function parameters that are consistent with the provided type context to ensure type -correct and semantically meaningful test inputs.
Assertions: Ensure that the assertions are meaningful and correctly validate the function's expected behavior.
Edge Cases: Consider different input scenarios, including edge cases and potential failure points.

Function Code:
Module: {**Module Path**}
"""
{**Summary**}
{**Type Context**}
"""
{**Source Code**}

Fig. 4. The core part of the prompt for test case generation.

3.4.2 Adaptive Error Repair. In this study, we adopt a regression-oriented design, where the observable runtime behavior of the current program version is treated as semantically correct. Accordingly, any generated test whose execution outcome contradicts this behavior is regarded as

inconsistent. Despite careful prompt engineering, the generated test case t may still contain both syntactic defects and semantic inconsistencies. These errors exhibit heterogeneous characteristics.

Syntactic defects are predominantly manifested as `ModuleNotFoundError`, typically occurring when the LLM fails to correctly resolve the class dependencies of the function under test or its parameters. Semantic inconsistencies are more prevalent, among which `AssertionError` and `AttributeError` are the most frequently observed. An `AssertionError` usually results from insufficient semantic information in the prompt or limitations in the LLM's reasoning capability. In contrast, an `AttributeError` often arises when an improperly inferred object type causes invalid member access.

The heterogeneity and unpredictability of these error types render static, rule-based repair strategies insufficient. To address this limitation, we design an *adaptive error repair mechanism* that leverages the LLM's context-sensitive reasoning capability, complemented by external knowledge retrieval, to iteratively diagnose and rectify faults. The mechanism consists of four stages:

(1) **Context-Aware Diagnosis.** LLM analyzes the error trace, localizes the faulty component, and generates candidate rectification strategies.
(2) **External Knowledge Augmentation.** Building on the class-level retrieval framework in Section 3.3, we extend it to the function level. The function summaries are embedded as indexing keys in a vector database, over which LLM-generated queries are executed to retrieve supplementary contextual information when necessary.
(3) **Repair Synthesis.** Guided by the diagnostic and retrieval results, LLM synthesizes a refined version of the test case that resolves the identified fault.
(4) **Iterative Validation and Reduction.** The repaired test case is executed and, if errors persist, the cycle repeats. Once the number of repair attempts exceeds a predefined threshold, a heuristic reduction strategy progressively minimizes the test case until an executable and error-free variant is obtained.

This adaptive repair strategy equips TEST4PY with resilience against diverse and unforeseen error patterns, thereby substantially improving the robustness and executability of LLM-generated test cases.

4 Evaluation

In this section, we evaluate TEST4PY by addressing the following research questions:

RQ1: How does TEST4PY compare with state-of-the-art baselines in terms of code coverage?
RQ2: How do different large language models affect the performance of TEST4PY in test generation?
RQ3: How effective is TEST4PY 's type inference module compared to existing type inference tools, and what is its impact through ablation analysis?
RQ4: How does incorporating call graph-guided summaries influence the effectiveness of test case generation in TEST4PY?
RQ5: How does error repair and iterative test case generation improve the quality of test suites produced by TEST4PY?

Benchmarks. We conduct experiments on two complementary benchmarks. The first is the benchmark provided by Pynguin [22], hereafter referred to as **Pyn**. This benchmark has been widely adopted by tools such as CodaMosa [19] and CoverUp [1]. **Pyn** consists of 17 real-world Python projects drawn from datasets including BugsInPy [47] and ManyTypes4Py [27]. All modules in this dataset contain type hints compliant with PEP 484[4], which defines the standard for type

[4]https://peps.python.org/pep-0484/

Table 1. Overview of the NA Benchmark: Nine open-source Python projects without type annotations, collected from GitHub.

Repository	Commit	Modules	Lines	Branches
mindsdb/dfsql	2c98482	10	992	383
box/genty	85f7c96	4	204	72
devshawn/kafka-shell	3615895	11	595	170
vitsalis/PyCG	8d5dc40	20	2032	961
kieferk/pymssa	9d4d3e2	3	413	146
skelsec/pypykatz_server	bdc76f4	5	258	48
tobgu/pyrthon	a874549	2	130	33
btwael/superstring.py	f47ef78	1	136	34
docwza/woa	4747379	2	112	24

annotations in Python. To avoid trivial and untestable modules, we refer to the filtered benchmark provided by CodaMosa [19], which results in a subset of 125 modules.

To complement this type-annotated benchmark, we construct a second dataset, denoted as **NA**, comprising nine open-source projects with more than 50 GitHub stars but no type annotations. We employed AST analysis to select projects in which fewer than 5% of function parameters contained type annotations. Furthermore, to mitigate the risk of data leakage, we filtered out projects in which the source code and pre-existing test cases were co-located within the same directory. As shown in Table 1, these projects contain 58 modules in total. Together, **Pyn** and **NA** allow us to assess the performance of test generation tools in both type-rich and type-scarce environments.

Baselines and Experimental Setup. We compare TEST4PY against two state-of-the-art LLM-based unit test generation tools: CodaMosa [19] and CoverUp [1]. CodaMosa integrates large language models with search-based testing, iteratively generating new test cases to overcome coverage bottlenecks. CoverUp adopts a complementary strategy by combining LLMs with coverage-guided feedback to generate high-coverage regression tests.

To ensure fair evaluation, we align experimental configurations across tools:

- For TEST4PY, we adopt a retrieval-augmented generation configuration, using the "BAAI/bge-large-en-v1.5" model for embeddings[5], and Chroma[6] for building behavior-based indices.
- We use `gpt-4o` as the default underlying LLM for all tools in RQ1. Unless otherwise specified, all other experiments use `deepseek`. TEST4PY and CoverUp are configured to perform three rounds of test generation, producing one test per function in each round.
- For CodaMosa, we follow the original protocol, executing three rounds of testing with a 10-minute budget per module. Since the original CodaMosa employed Codex [6], which produces code-completion style outputs, we adapted its output parsing to align with conversational LLMs, ensuring comparability across tools.

4.1 RQ1: Effectiveness of TEST4PY

This research question investigates the effectiveness of TEST4PY in automated test generation. Specifically, we examine whether TEST4PY achieves higher test coverage than existing state-of-the-art approaches and whether it maintains stable performance across different benchmarks. To this

[5]https://huggingface.co/BAAI/bge-large-en-v1.5
[6]https://github.com/chroma-core/chroma

end, we conduct quantitative comparisons with CoverUp and CodaMosa using standard coverage metrics and further complement the analysis with a qualitative case study.

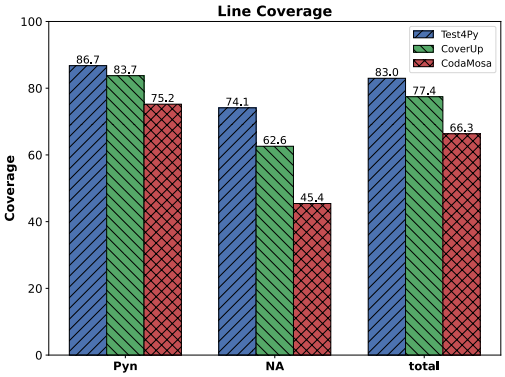

Fig. 5. Line Coverage Comparison

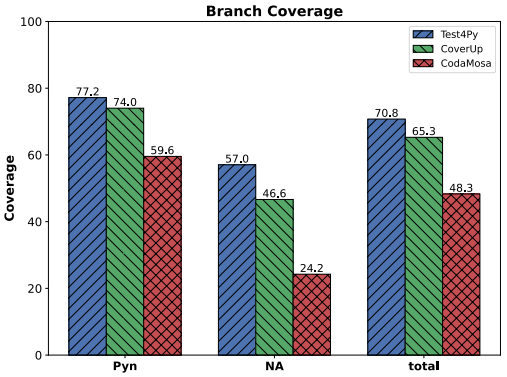

Fig. 6. Branch Coverage Comparison

Test Coverage. Following the evaluation methodology of CoverUp, we measure both line coverage and branch coverage across benchmarks. Figures 5 and 6 report the results. On average, Test4Py attained the highest line coverage (83.0%), surpassing CoverUp (77.4%) and CodaMosa (66.3%). Similarly, it achieved the highest branch coverage (70.8%), outperforming CoverUp (65.3%) and CodaMosa (48.3%). These results demonstrate that Test4Py consistently generates test suites that achieve broader exploration of program behaviors.

A closer inspection reveals different patterns across datasets. On the **Pyn** benchmark, the gap between Test4Py and CoverUp was relatively small. This is largely attributed to CoverUp's mechanism of dynamically retrieving function or class definitions through function calls, which is particularly effective in **Pyn** where most parameters are annotated with type hints. However, in the **NA** benchmark, which lacks type hints, CoverUp's mechanism becomes less effective, leading to a substantial performance decline. Consequently, in **NA**, Test4Py outperformed CoverUp by 18.4 percentage in line coverage and 22.3 in branch coverage, compared to smaller differences of 7.2 and 8.4 when averaged across both benchmarks.

To further validate these observations, we conduct paired Wilcoxon signed-rank tests to assess whether the coverage differences between tools are statistically significant on the same modules. In the **NA** benchmark, Test4Py significantly outperforms both CoverUp ($p = 0.0004$) and CodaMosa ($p < 0.0001$). The corresponding effect sizes are $\hat{A}_{12} = 0.5815$ (vs. CoverUp) and $\hat{A}_{12} = 0.6650$ (vs. CodaMosa), indicating small-to-medium practical effects. In the **Pyn** benchmark, although the performance gap is smaller, Test4Py still shows statistically significant differences in paired comparisons ($p = 0.0367$ vs. CoverUp; $p < 0.0001$ vs. CodaMosa), with effect sizes of $\hat{A}_{12} = 0.5236$ and $\hat{A}_{12} = 0.6382$, respectively. These results confirm that Test4Py is particularly advantageous when type hints are unavailable, while remaining competitive when type information is provided.

Stability of Coverage. To evaluate performance stability, we analyzed the distribution of the line coverage at the module level using box plots, as shown in Figure 7 and 8. The median (Q2) indicates the central trend, while the interquartile range (IQR) reflects the variability between modules. A smaller IQR indicates more stable performance.

On the **Pyn** benchmark, Test4Py achieved a median coverage of 95.7, slightly higher than CoverUp (94.4). More importantly, its IQR (17.7) was considerably lower than that of CoverUp (21.7)

and CodaMosa (40.2), reflecting its robustness across diverse modules. On the **NA** benchmark, Test4Py exhibited even stronger advantages, with a median coverage of 92.2, exceeding CoverUp (79.1) and CodaMosa (69.3). Its IQR (26.5) was the smallest among all tools, representing only 52.5% of CoverUp's and 38.5% of CodaMosa's. These results confirm that Test4Py achieves not only higher coverage but also more consistent performance across modules.

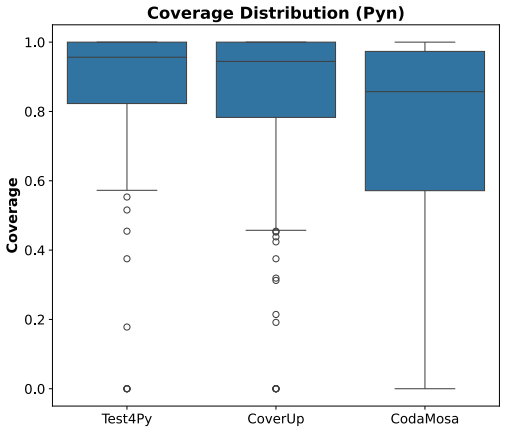

Fig. 7. Coverage Distribution on Pyn Dataset

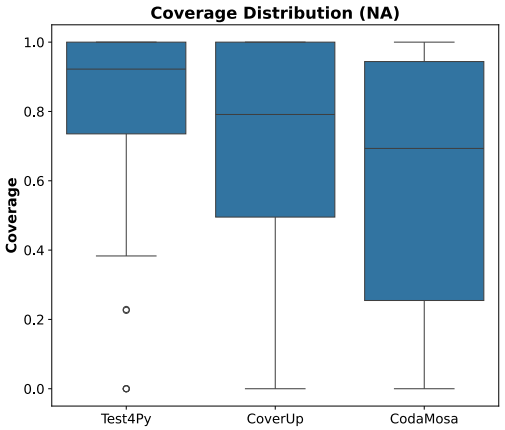

Fig. 8. Coverage Distribution on NA Dataset

Case Study. To evaluate the effectiveness of Test4Py in detecting real-world regression bugs, we conducted a case study on GitHub pull requests (PRs). We first collected PRs explicitly labeled as regression bugs. To ensure the availability of ground-truth regression information, we applied a validation process combining an LLM and three expert PhD students to retain only those PRs that clearly specify the version in which the regression was introduced.

For each selected PR, we retrieved two versions of the codebase: (1) the pre-bug version (v_1) and (2) the post-bug version (v_2). We then identified the functions modified in each PR and defined them as *target functions*. To ensure validity, we further excluded cases where target functions were not analyzable, such as modifications involving non-Python files or functions that do not exist in v_1. After this process, we obtained 18 valid instances spanning 8 different repositories, including widely used projects such as *sympy* and *pylint*.

To assess regression detection capability, we adopt a two-stage evaluation protocol. The first stage uses the *pass-to-fail* criterion, where a generated test case passes on v_1 but fails on v_2, indicating potential regression detection. However, such failures may also result from intentional changes (e.g., interface modifications) rather than actual regression faults. To address this ambiguity, the second stage performs semantic validation using both an LLM-as-a-judge and human experts. Based on the PR descriptions, the LLM and the PhD students independently assess whether each pass-to-fail test case corresponds to the reported regression bug. All evaluated tools are required to generate test cases targeting the identified target functions.

We compared Test4Py with CoverUp. To control computational overhead, we restricted Test4Py 's analysis scope, as it is designed for project-level test generation whereas our evaluation targets individual functions. Specifically, for each instance, we limited Test4Py to at most 100 summaries propagated through the call graph. In contrast, CoverUp discards test cases that do not improve coverage, although such tests may still expose regression faults; therefore, we retained them in our evaluation.

The experimental results indicate that Test4Py generated pass-to-fail test cases for 6 out of 18 instances, whereas CoverUp achieved this for only 2 instances. After validation by both the LLM and human experts, 5 of the test cases produced by Test4Py were confirmed to be genuinely associated with regression bugs, with only one case attributed to intended functional changes. In contrast, CoverUp produced 2 validated regression-revealing test cases.

Further analysis reveals that this performance gap can be attributed to two primary factors. First, CoverUp frequently generates overly trivial test cases (e.g., PR #274 in *sarracenia*) or fails to produce valid passing test cases on the previous version (e.g., PR #11353 in *yt-dlp*), thereby limiting its ability to expose behavioral differences across versions. Second, while Test4Py may miss relevant code regions in some cases (e.g., PR #9712 in *yt-dlp*), leading to undetected regressions, it produces more discriminative test cases when successful. These tests more effectively capture regression-inducing changes, demonstrating that Test4Py exhibits stronger capability in regression detection.

Summary: Test4Py consistently achieves higher and more stable coverage than state-of-the-art baselines. Its advantages are especially pronounced in scenarios without type hints, underscoring its effectiveness in dynamically typed settings.

4.2 RQ2: The Impact of Different Large Language Models on Test4Py

This research question examines how the choice of large language model (LLM) influences the effectiveness of Test4Py. In particular, we aim to assess whether different LLMs exhibit varying levels of test generation capability.

We evaluated Test4Py using three LLMs: `gpt-4o-2024-05-13`, `deepseek-v3-250324`, and `qwen3-32b` (hereafter referred to as *gpt-4o*, *deepseek*, and *qwen*, respectively). The models were tested on both **Pyn** and **NA**, and their performance was measured in terms of line and branch coverage. Figure 9 presents the results in a radar chart, with six dimensions corresponding to line and branch coverage across the two datasets.

Overall, *deepseek* achieved the highest performance, with an average line coverage of 85.1% and branch coverage of 74.2%, consistently outperforming the other models across all dimensions. *gpt-4o* ranked second, showing competitive performance that was close to *deepseek* on the **Pyn** benchmark but lagging more substantially on the **NA** benchmark. Due to its relatively smaller parameter scale, *qwen* yielded the lowest performance among the three models. Nevertheless, it still attained an average line coverage of 71.3%, indicating that Test4Py remains effective even when deployed with comparatively small-scale models.

We further investigate the impact of different underlying LLMs on CoverUp, with the results presented in Figure 10. Across all models and benchmarks, Test4Py consistently outperforms CoverUp. CoverUp achieves its highest performance with *gpt-4o*, but its branch coverage decreases by 7.9% when switching to *deepseek*. In contrast, Test4Py exhibits a 4.8% performance gain when using *deepseek* compared to *gpt-4o*. These findings indicate that Test4Py maintains robust performance despite variations in the quality of the underlying model.

When using the *qwen* model, Test4Py incurs an average cost of only $1.10 per project, whereas CoverUp requires approximately $0.28 per project on average. Considering the critical role of test generation throughout the software development lifecycle, an expenditure of $1.10 per project remains entirely acceptable. In this context, the additional cost constitutes a reasonable trade-off for achieving higher coverage and improved test suite quality.

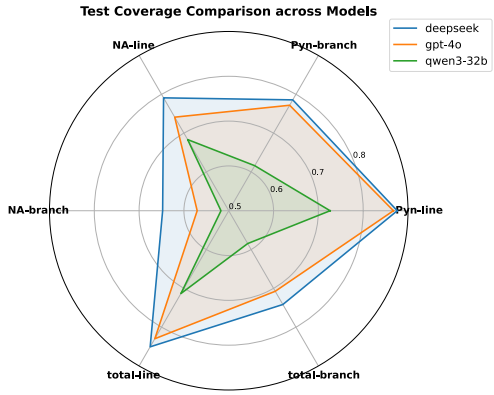

Fig. 9. Line and Branch Coverage of Different LLMs across Benchmarks

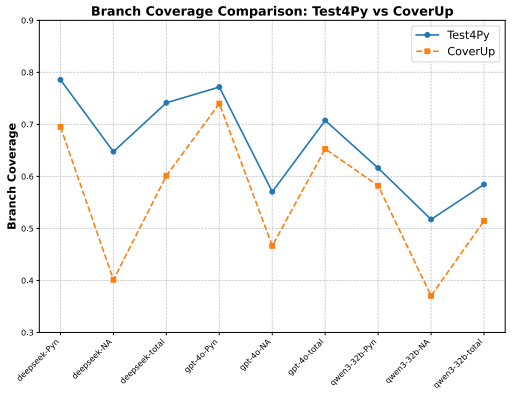

Fig. 10. Comparison of Branch Coverage between Test4Py and CoverUp across Different Models

Summary: Different LLMs have a measurable impact on the performance of Test4Py. *Deepseek* demonstrates the highest coverage, followed by *gpt-4o* and *qwen*. We further evaluated state-of-the-art baselines across the same models and observed that Test4Py exhibits greater stability across model variations.

4.3 RQ3: The Effectiveness and Ablation of Test4Py's Type Inference

This research question evaluates the performance of Test4Py's type inference process compared to state-of-the-art (SOTA) tools and presents an ablation study of the type inference component.

Numerous Python type-inference tools have been developed, such as Type4Py [28], HiTyper [34] and TypeGen [35]. Among these, HiTyper effectively integrates static inference with deep learning techniques. Recently, with the advancement of LLMs, their type inference performance has even surpassed that of HiTyper [35]. Therefore, we chose Type4Py, HiTyper and TypeGen baseline for our comparison. To further investigate the individual contributions of syntactic filtering and semantic retrieval in our framework, we conducted ablation studies. Specifically, we denote the variant that applies only syntactic filtering as Test4Py-sy, and the variant that employs only semantic retrieval as Test4Py-se.

The **Pyn** benchmark contains a substantial number of type annotations. We removed these function parameter annotations and treated them as the ground truth, thus creating the **Pyn-type** benchmark. This benchmark includes 222 User-defined types and 1806 Non-user-defined types. We define an Exact Match as a strict match and a Relaxed Match as a loose match (e.g., for List[int], only the outermost List type is checked). We ran the six aforementioned tools on the **Pyn-type** benchmark, and the results are presented in Table 2.

As shown in the results, Test4Py achieves the highest prediction performance among all evaluated approaches. Its accuracy on user-defined types reaches 63.5%, substantially outperforming the baseline TypeGen, which reaches 19.8%. For non-user-defined types, Test4Py achieves an accuracy of 68.1%, which is comparable to TypeGen's 58.2%. In contrast, Type4Py demonstrates particularly weak performance on user-defined types, suggesting that traditional machine learning-based approaches are not well-suited for this task. HiTyper exhibits consistently poor performance across all categories, primarily due to its inability to handle certain language constructs (e.g., "for loops with else statements"), which prevents it from producing valid results. These findings highlight the

Table 2. Type Prediction Results: Relaxed vs. Exact Matching, split by User-defined and Non-user-defined types.

Setting	Type Category	Relaxed Match		Exact Match	
		Success	Failure	Success	Failure
TEST4PY	User-defined	**141 (63.5%)**	81 (36.5%)	**81 (36.5%)**	141 (63.5%)
	Non-user-defined	1230 (68.1%)	576 (31.9%)	**1006 (55.7%)**	800 (44.3%)
TEST4PY-SE	User-defined	137 (61.7%)	85 (38.3%)	80 (36.0%)	142 (64.0%)
	Non-user-defined	**1238 (68.5%)**	568 (31.5%)	1001 (55.4%)	805 (44.6%)
TEST4PY-SY	User-defined	117 (52.7%)	105 (47.3%)	67 (30.2%)	155 (69.8%)
	Non-user-defined	1222 (67.7%)	584 (32.3%)	992 (54.9%)	814 (45.1%)
TypeGen	User-defined	44 (19.8%)	178 (80.2%)	30 (13.5%)	192 (86.5%)
	Non-user-defined	1051 (58.2%)	755 (41.8%)	928 (51.4%)	878 (48.6%)
Type4Py	User-defined	31 (14.0%)	191 (86.0%)	23 (10.4%)	199 (89.6%)
	Non-user-defined	852 (47.2%)	954 (52.8%)	751 (41.6%)	1055 (58.4%)
HiTyper	User-defined	11 (5.0%)	211 (95.0%)	8 (3.6%)	214 (96.4%)
	Non-user-defined	452 (25.0%)	1354 (75.0%)	412 (22.8%)	1394 (77.2%)

limitations of conventional type inference tools and underscore the robustness and effectiveness of TEST4PY 's inference mechanism.

The precision of TEST4PY-se is comparable to that of TEST4PY, indicating that the semantic retrieval mechanism achieves high precision, accurately identifying the correct type in most cases even without syntactic filtering. However, this does not imply that syntactic filtering is ineffective. In contrast, it significantly reduces the computational cost of retrieval. Specifically, when syntactic filtering is disabled, each query produces an average of 41.38 candidate classes; incorporating syntactic filtering reduces this number to 28.74, achieving a reduction of 30.5%. Although TEST4PY-sy exhibits a noticeable performance drop compared to TEST4PY, it still substantially outperforms the TypeGen baseline, further confirming the effectiveness of syntactic filtering.

To further investigate the contribution of TEST4PY 's type inference to test case generation, we conducted an ablation study. Specifically, we implemented a variant, *NoTypeInfer*, which disables the type inference component. Both versions were evaluated on the **NA** benchmark, and the differences in branch coverage are illustrated in Figure 11. Compared with *NoTypeInfer*, TEST4PY achieved higher coverage in 25 of the 58 modules and similar coverage in 29 modules. We conducted a detailed analysis of the remaining four modules where coverage decreased. Three of these degradations can be attributed to the inherent nondeterminism of LLM outputs, while one case was likely due to erroneous type inference

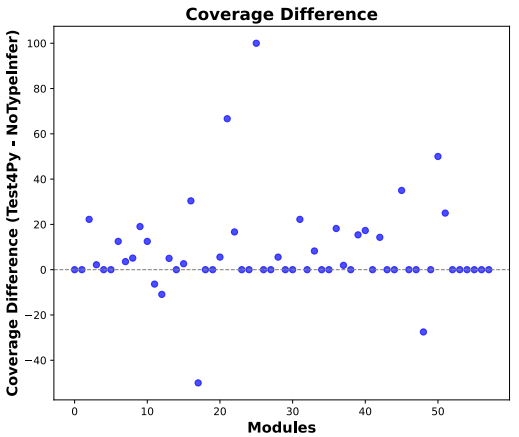

Fig. 11. Coverage difference with and without TEST4PY's type inference module

introducing misleading contextual cues into the prompt. The most pronounced difference was observed in the `pymssa.ops` module. For the `decompose_trajectory_matrix` function, Test4Py generated input parameters using *np.random*, which resulted in execution failures. In contrast, *NoTypeInfer* generated parameters using *np.array*, thereby producing valid inputs and achieving higher coverage for this function.

Although the type inference process still exhibits a relatively high error rate, Test4Py mitigates its negative effects by leveraging the LLM's capacity to filter irrelevant information, thereby limiting adverse impact to a small minority of cases. Overall, these findings demonstrate that type inference consistently enhances the effectiveness of Test4Py in scenarios where type annotations are absent.

In addition to coverage metrics, we further assess the contribution of type inference through the case study described in Section 4.1. The results show that *NoTypeInfer* triggered four pass-to-fail transitions, of which two were confirmed as regression-revealing tests, compared with Test4Py, which triggered six pass-to-fail transitions, five of which were confirmed as regression-revealing. Without parameter type information, generated tests may crash due to type errors unrelated to the function logic (e.g., PR #274 in *sarracenia*), and can produce semantically trivial parameters that fail to exercise certain execution paths (e.g., PR #11353 in *yt-dlp*). These observations suggest that integrating type inference not only improves coverage but also reduces semantically invalid or trivial tests, supporting more meaningful regression detection.

Summary: Test4Py's type inference is more effective than HiTyper and TypeGen, especially when handling user-defined types. Additionally, the type inference system enables Test4Py to achieve better performance in the absence of type annotations.

4.4 RQ4: The Role of Call Graph-Based Summaries

This research question investigates the effectiveness of incorporating call graph-guided information into function summaries for test case generation. Specifically, we evaluate whether interprocedural context captured by call graph analysis improves the semantic adequacy and fault-detection capability of the generated test cases. To this end, we conducted an ablation study by creating a variant, NoGraph, in which the prompts exclude any contextual information derived from call graph analysis. We compared Test4Py and NoGraph on two benchmarks in terms of branch coverage. Test4Py achieved 74.2% coverage, while NoGraph reached 72.6%.

By incorporating interprocedural semantic context, these summaries guide the LLM to generate test cases that exercise more diverse and semantically meaningful behaviors. For instance, in the *dataclasses_json.utils* module, NoGraph predominantly produced trivial input-output identity checks, whereas Test4Py generated test cases that better reflected realistic usage scenarios of the target function.

However, since coverage metrics cannot effectively distinguish trivial tests from behaviorally meaningful ones, we further validate the effectiveness of our approach through the case study presented in Section 4.1.

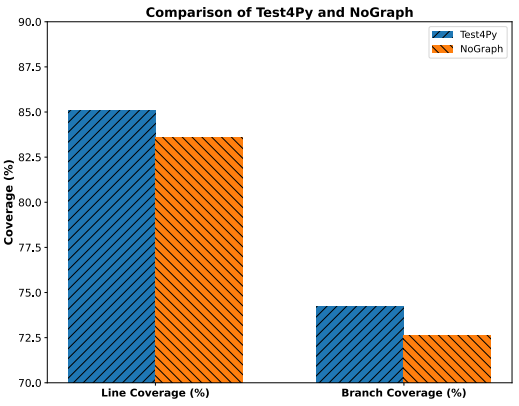

Fig. 12. Line coverage and branch coverage of Test4Py and NoGraph

The results show that NoGraph triggered five pass-to-fail transitions, but only two of them correspond to genuinely regression-revealing tests. Without call graph-based summarization, generated tests tend to focus on trivial functionalities and exhibit weaker alignment with real-world usage semantics. This leads to a substantial number of non-regression-revealing cases (e.g., PR #274 in *sarracenia*), despite achieving high coverage. These findings indicate that call graph-based summarization improves the precision of regression bug detection.

Summary: Call graph-guided summaries improve both line coverage and branch coverage, and enhance the semantic adequacy of the generated test cases.

4.5 RQ5: Test Case Repair and Iterative Generation

This research question examines the role of automated repair and the effectiveness of iterative test case generation in improving the overall quality of the generated test suite. Specifically, we analyze (i) the types of errors encountered and the effectiveness of the repair mechanism, and (ii) the distribution of execution time across different stages of test case generation.

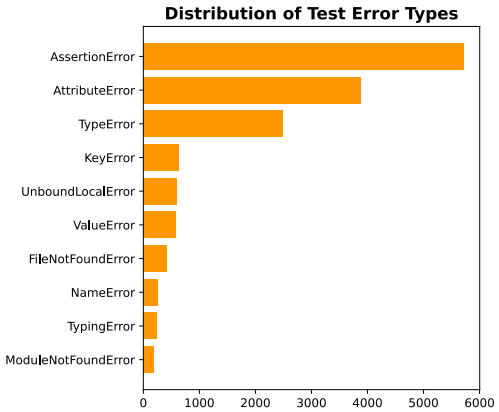

Fig. 13. Distribution of the ten most frequent error types.

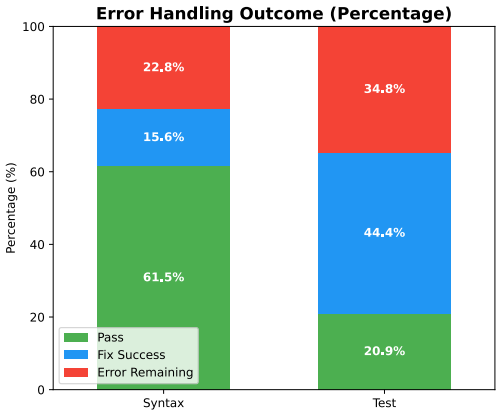

Fig. 14. Error handling outcomes for syntax and test errors.

Error Characterization and Repair Effectiveness. We analyzed the top-10 error types observed in the generated tests (Figure 13). The most frequent error is `AssertionError`, highlighting the difficulty LLMs face in generating precise and semantically valid assertions [56]. The next most common errors, `AttributeError` and `TypeError`, indicate persistent challenges related to type inference and type consistency. Despite recent progress in type-aware generation, type-related issues remain a key bottleneck.

Repair outcomes are summarized in Figure 14. For syntax errors, the initial correctness rate was 61.5%, and among the remaining erroneous tests, 40.6% were successfully repaired. For semantic test errors, only 20.9% of the tests passed initially, but the repair mechanism successfully resolved 56.1% of the failing tests. The lower initial pass rate is largely attributable to LLM's strategy of

generating multiple assertions per test, which decreases the probability of immediate success. Nevertheless, iterative repair effectively transforms failing test cases into passing ones, demonstrating the robustness of the repair module.

Table 3. Execution time distribution and coverage improvement across generations.

Stage	Time Proportion	Coverage Achieved
Summarization	27.6%	–
Generation 1	40.4%	0.652
Generation 2	17.5%	0.712
Generation 3	14.5%	0.742

Execution Time and Coverage Evolution. Table 3 reports the relative time cost of summarization and each generation phase, together with the coverage achieved. The first generation dominates execution time since TEST4PY must generate tests for all uncovered functions. In subsequent iterations, functions that have already achieved full coverage are excluded, resulting in progressively shorter generation phases. Importantly, coverage consistently improves across iterations (from 65.2% to 74.2%), suggesting that iterative generation can substantially increase the achievable coverage. Removing the limit on the number of iterations may therefore yield even higher coverage.

Summary: Automated repair substantially mitigates both syntax- and semantics-related errors, ensuring that the final test suite is both comprehensive and reliable.

5 Threats to Validity

The broader challenge of data contamination remains a fundamental concern for the reliability of evaluating LLM-based testing frameworks. To mitigate this risk, future work will focus on developing a continually updated benchmark dataset that minimizes contamination and ensures the robustness and fairness of empirical evaluations. Moreover, TEST4PY currently lacks specialized handling for less prevalent third-party libraries, those that do not appear in the target project and are unlikely to be represented in the LLM's training data. This limitation highlights a promising direction for further improvement, and future research will extend TEST4PY to better support such underrepresented libraries.

6 Related Work

Our work relates to the following areas: Search-Based Software Testing (SBST), and LLM-based unit test generation.

6.1 Search Based Software Testing (SBST)

SBST employs search algorithms to automatically generate test cases [14, 39, 44], which can greatly reduce the time developers spend on test case creation [41], while also generating boundary cases and exceptional inputs that are often challenging to identify manually. Various SBST-based tools and algorithms have been developed to generate test cases for programming languages such as Java, Python, and JavaScript, some of which support multiple testing objectives, including line and branch coverage [10, 12, 17, 29].

EvoSuite [20, 42, 43] is a well-known tool based on SBST, which automatically generates JUnit test suites that maximize code coverage. Randoop [32] is a feedback-directed random test generation tool that generates test cases by randomly combining previously executed statements that did not result in failures. Algorithm MOSA (Multi-Objective Simulated Annealing) [40], and algorithms related to it, such as DynaMOSA [33] and MIO (Many Independent Objective) [3, 4], are used for test generation, and particularly effective at handling multiple objectives, containing line coverage, branch coverage, and multiple mutants in mutation testing. Sapienz [5, 25, 30] is an approach to Android testing that uses multi-objective SBST to optimize test sequences for brevity and effectiveness in revealing faults. Sapienz leverages a combination of random fuzzing, systematic exploration, and SBST. Pynguin is an extendable test-generation framework for Python, which is a dynamically typed programming language [16, 22, 23, 36].

Despite the development of many SBST-based test case generation tools, they have notable limitations. These tools typically produce only boundary condition test cases, with assertions limited to simple equality checks. Additionally, when software updates occur, generating new test cases can be time-consuming, even for minimal changes. Furthermore, inaccurate type inference restricts their effectiveness, particularly in dynamically-typed languages like Python, which employs duck typing [23]. In contrast, TEST4PY generates test cases providing more accurate type inference, enhancing overall test effectiveness.

6.2 LLM-Based Unit Test Generation

LLM-based unit test generation [38] is the technique that leverages large language models trained to automatically generate unit tests for software code. There has been a large number of works or tools generating test cases with Large Language Models (LLMs) [48, 50, 52], demonstrating impressive results. MuTAP [8] is a prompt-based learning technique to generate effective test cases with LLMs, which improves the effectiveness of test cases generated by LLMs in terms of revealing bugs by leveraging mutation testing. TestPilot [38] is a tool for automatically generating unit tests for npm packages written in JavaScript/TypeScript using LLM, which provides LLM with the signature and implementation of the function under test, along with usage extracted from the documentation. ChatUniTest [7] utilizes an LLM-based approach encompassing valuable context in prompts and rectifying errors in generated unit tests. ChatTester [55] is a Maven plugin similar to ChatUniTest above, which leverages ChatGPT to improve the quality of its generated tests. LLM4Fin [49] is designed for testing real-world stock-trading software, which generates comprehensive testable scenarios for extracted business rules. CodaMosa [19] developed by Microsoft combines SBST and LLMs, using LLM to help SBST's exploration. LLM will provide examples for SBST when its coverage improvements stall to help SBST search more useful areas. LegaTest [13] further integrates LLMs with genetic algorithms to improve coverage, executability, and assertion quality in unit test generation. Recent work [56] explores automated assertion generation with LLMs, conducting a large-scale evaluation of different models and demonstrating their effectiveness in improving assertion quality and detecting real-world bugs. Beyond unit test generation, MASFuzzer [21] leverages LLMs to generate fuzz drivers from multidimensional API sequences and uses coverage-guided scheduling to improve library fuzzing effectiveness.

While existing tools rely on prompts to guide LLMs in generating test cases, they fail to incorporate type information. For instance, TestPilot does not adapt its prompts based on type information, nor does it refine them when type information does not improve. Similarly, CodaMosa only prompts the LLM when necessary to support SBST, positioning the LLM as a supplementary component rather than a primary driver of test case generation. Type inference has achieved considerable success [51], yet its potential has not been effectively exploited in LLM-based test generation. Pytlm [54] assists SBST through inferred type information, yet does not leverage user-defined type

information to improve test generation. In contrast, our tool, TEST4PY, leverages type information to guide the LLM in generating test cases that are more likely to improve coverage, thereby enriching the LLM with additional, type-aware information.

7 Conclusion

We propose a type-aware approach to automated test generation, designed to enhance the validity and correctness of test cases for Python programs by inferring and incorporating precise type information during test construction. TEST4PY leverages call graph-guided analysis to more effectively capture the semantic context of function parameters. Furthermore, by employing behavior-guided parameter type inference, TEST4PY improves the accuracy of parameter type prediction and strengthens the type soundness of generated test inputs. To ensure robustness, TEST4PY integrates an automated fault repair mechanism, enabling the production of more comprehensive and reliable test suites. We conducted an extensive empirical evaluation across 183 real-world Python modules. The results show that TEST4PY consistently outperforms state-of-the-art baselines in both the effectiveness and stability of the generated test cases. Future work will focus on extending the applicability of TEST4PY to other dynamically typed languages and further improving its efficiency.

Acknowledgments

This work was supported by the National Key R&D Program of China No. 2024YFB4506200.

References

[1] Juan Altmayer Pizzorno and Emery D Berger. 2025. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2897–2919.

[2] J. H. Andrews, T. Menzies, and F. C. H. Li. 2011. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering* 37, 1 (2011), 80–94.

[3] Andrea Arcuri. 2017. Many independent objective (MIO) algorithm for test suite generation. In *International symposium on search based software engineering*. Springer, 3–17.

[4] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206.

[5] Iván Arcuschin, Juan Pablo Galeotti, and Diego Garbervetsky. 2023. An Empirical Study on How Sapienz Achieves Coverage and Crash Detection. *Journal of Software: Evolution and Process* 35, 4 (2023), e2411.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.

[8] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468.

[9] Matthew C Davis, Sangheon Choi, Sam Estep, Brad A Myers, and Joshua Sunshine. 2023. NaNofuzz: A Usable Tool for Automatic Test Generation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1114–1126.

[10] Pouria Derakhshanfar and Xavier Devroey. 2022. Basic block coverage for unit test generation at the SBST 2022 tool competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing*. 37–38.

[11] Matouš Eibich, Shivay Nagpal, and Alexander Fred-Ojala. 2024. ARAGOG: Advanced RAG output grading. *arXiv preprint arXiv:2404.01037* (2024).

[12] Raihana Ferdous, Chia-kang Hung, Fitsum Kifetew, Davide Prandi, and Angelo Susi. 2022. EvoMBT at the SBST 2022 tool competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing*. 51–52.

[13] Yiwen Fu, Xiang Gao, Binhang Qi, Yuan Yuan, and Hailong Sun. 2026. Fusing LLMs and Genetic Algorithm for High-Quality Unit Test Generation. *ACM Transactions on Software Engineering and Methodology* (2026).

[14] Sepideh Kashefi Gargari and Mohammd Reza Keyvanpour. 2021. SBST challenges from the perspective of the test techniques. In *2021 12th International Conference on Information and Knowledge Technology (IKT)*. IEEE, 119–123.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.
[16] Lucca Guerino and Auri Vincenzi. 2023. An Experimental Study Evaluating Cost, Adequacy, and Effectiveness of Pynguin's Test Sets. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*. 5–14.
[17] Giovani Guizzo and Sebastiano Panichella. 2023. Fuzzing vs sbst: Intersections & differences. *ACM SIGSOFT Software Engineering Notes* 48, 1 (2023), 105–107.
[18] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. Justgen: Effective test generation for unspecified JNI behaviors on jvms. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1708–1718.
[19] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.
[20] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1068–1080.
[21] Xingyu Liu, Zengqin Huang, Xiang Gao, and Hailong Sun. 2026. MASFuzzer: Fuzz driver generation and adaptive scheduling via multidimensional api sequences. *arXiv preprint arXiv:2604.17977* (2026).
[22] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.
[23] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* 28, 2 (2023), 36.
[24] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. Grt: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 212–223.
[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*. 94–105.
[26] Yuning Mao, Yanru Qu, Yiqing Xie, Xiang Ren, and Jiawei Han. 2020. Multi-document summarization with maximal marginal relevance-guided reinforcement learning. *arXiv preprint arXiv:2010.00117* (2020).
[27] 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.
[28] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2241–2252.
[29] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. 2021. Deeper at the sbst 2021 tool competition: Adas testing using multi-objective search. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 40–41.
[30] Iván Arcuschin Moreno, Juan Pablo Galeotti, and Diego Garbervetsky. 2020. Algorithm or Representation? An empirical study on how SAPIENZ achieves coverage. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 61–70.
[31] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 922–934.
[32] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
[33] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.
[34] 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.
[35] 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)*. IEEE, 988–999.
[36] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Cristina Seceleanu, Wasif Afzal, and Filip Sebek. 2023. Automating test generation of industrial control software through a plc-to-python translation framework and pynguin. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 431–440.
[37] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE,

1646–1657.

[38] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[39] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1340–1359.

[40] Ekunda Lukata Ulungu, JFPH Teghem, PH Fortemps, and Daniel Tuyttens. 1999. MOSA method: a tool for solving multiobjective combinatorial optimization problems. *Journal of multicriteria decision analysis* 8, 4 (1999), 221.

[41] MS Vasudevan, Santosh Biswas, and Aryabartta Sahu. 2021. Automated low-cost sbst optimization techniques for processor testing. In *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*. IEEE, 299–304.

[42] Sebastian Vogl, Sebastian Schweikl, and Gordon Fraser. 2021. Encoding the certainty of boolean variables to improve the guidance for search-based test generation. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1088–1096.

[43] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. 2021. EVOSUITE at the SBST 2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 28–29.

[44] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

[45] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[46] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[47] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

[48] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. Mr-adopt: Automatic deduction of input transformation function for metamorphic testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 557–569.

[49] Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024. Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1643–1655.

[50] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing llm-based test generation for hard-to-cover branches via program analysis. *arXiv e-prints* (2024), arXiv–2404.

[51] Guowei Yang, Shilin He, Fu Song, and Yuqi Chen. 2025. RunTyper: Enhancing Deep Type Inference Using Dynamic Analysis for Python. *ACM Transactions on Software Engineering and Methodology* (2025).

[52] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1607–1619.

[53] Qian Yang, J Jenny Li, and David Weiss. 2006. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*. 99–103.

[54] Ruofan Yang, Xianghua Xu, and Ran Wang. 2025. LLM-enhanced evolutionary test generation for untyped languages. *Automated Software Engineering* 32, 1 (2025), 20.

[55] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).

[56] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2025. Exploring automated assertion generation via large language models. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–25.

[57] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, and Yutian Tang. 2022. Selectively combining multiple coverage goals in search-based unit test generation. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*. 1–12.