Allegheny College
Computer And Information Science Department

Senior Thesis

Do Not Blame Me Mining and Mutating Type Checker Bugs; An LLM-Driven Approach to Differential Testing

by

Benedek Kaibas

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Project Supervisor: **Dr. Gregory M. Kapfhammer**
Co-Supervisor: **Professor Douglas Luman**

Abstract

Python's continued evolution and frequent releases of type checkers (e.g., mypy, pyrefly, and ty) create the need for a proactive verification technique. This paper introduces Pytifex, an automated differential testing agent that generates source code examples to expose edge cases and blind spots in Python type checkers. By using LLMs seeded with historical bug reports mined from GitHub repositories, Pytifex synthesizes known failures to generate novel code examples that surface contradictions among type checkers. These type checking disagreements reveal potential validity gaps and edge cases, aiding both the creators of type checkers and the software engineers who use them.

Table of contents

List of Figures

List of Tables

1 Introduction

By 2025, Python has established itself as the dominant programming language, a trend corroborated by major industry indices including the IEEE Spectrum and TIOBE [1, 2]. Its ecosystem is now foundational to diverse sectors, ranging from web and enterprise development to embedded systems and artificial intelligence. However, Python's flexibility comes at a cost: a proneness to runtime failures. Research indicates that type-related errors are among the most prevalent runtime exceptions in Python development [3]. These errors, which occur when an operation is performed on an incompatible data type, account for a significant portion of defects in open-source Python projects [4].

1.1 The Challange of Building and Maintaining Python Type Checkers

While the concept of static analysis is mature in languages like C, C++, and Java, implementing a robust type checker for Python presents unique and significant engineering challenges. Unlike statically typed languages where variable types are resolved at compile-time, Python is inherently dynamic. This flexibility, while beneficial for developer velocity, creates a hostile environment for static verification.

Runtime Metaprogramming and Reflection. Python allows extensive runtime manipulation through features like `setattr()`, `eval()`, and dynamic decorators. As Ren and Foster note, "prior approaches have trouble dealing with metaprogramming, which generates code as the program executes" [23]. This problem is particularly acute in frameworks like Django and Flask that rely heavily on such features. A type checker analyzing source code cannot "see" attributes or methods added at runtime, leading to false reports of missing attributes even when the code executes successfully.

Dynamic Language Features and Type Inference Limitations. Static type checkers are constrained by dynamic language features such as runtime type changes, reflection, monkey patching, and dynamic code generation [6, 20]. When encountering such features, checkers must conservatively over-approximate, often inferring `Any` types that reduce their ability to catch real bugs. Empirical studies confirm that reflection and introspection are pervasive in real-world Python codebases—Gao et al. found that 10.86% of functions across 35 popular projects used such features [21]—making this limitation practically significant.

The Any Type Escape Hatch. The `Any` type in Python's gradual type system acts as an escape hatch that disables type checking. Once a variable is inferred as `Any`—often due to missing annotations in third-party libraries or interaction with untyped code—the checker ceases to validate operations on it. This creates a blind spot where errors propagate silently through the codebase without detection. More broadly, analysis of 2,766 type error fixes from real-world Python projects confirms that type errors remain a persistent challenge even in annotated code [Chow et al. 2024].

Optional Annotations and Partial Coverage. Unlike statically typed languages such as Java or Rust, Python's type annotations are entirely optional and unchecked at runtime [28]. Developers add annotations incrementally, resulting in partially annotated codebases where some functions have complete type information while others have none [5]. When data flows from an untyped context into a typed context, the checker must make assumptions that can lead to both false positives (rejecting valid code) and false negatives (accepting type-incorrect code) [9].

These challenges are not merely implementation difficulties—they reflect fundamental

theoretical limitations in bringing static analysis to dynamic languages, which we explore in the next section.

1.2 The Complexity of Gradual Typing and Type Checker Soundness

Building a type checker for Python requires navigating fundamental theoretical limitations. Python implements gradual typing [26], a system that allows developers to incrementally add type annotations to their code. However, even when code is fully annotated, Python's dynamic runtime semantics create inherent challenges for static verification.

The core difficulty lies in a fundamental trade-off: gradual type systems intentionally sacrifice soundness for usability [28]. In type theory, a sound system guarantees that no ill-typed program is accepted (no false negatives), while a complete system ensures that no well-typed program is rejected (no false positives). Due to Python's highly dynamic features—such as runtime attribute modification, metaclass manipulation, and structural subtyping—achieving perfect soundness involves undecidable problems [16]. As a result, practical type checkers rely on approximations, and different checkers make different approximations—leading to the disagreements this work aims to expose.

1.3 Motivation

The Python typing ecosystem is experiencing a Cambrian explosion. Between mid-2024 and early 2025, the open-source community introduced several high-performance type checkers—pyrefly (Facebook), zuban, and ty (Astral, creators of ruff)—claiming execution speeds far superior to industry standards like mypy. Simultaneously, Python 3.12 and 3.13 introduced major new typing features, including PEP 695 type parameter syntax and PEP 742 TypeIs guards. This convergence creates a critical validation gap: new type checkers must correctly implement not only established features from a decade of PEPs (484, 544, 570, 586, 589, 591, 612, 647, 655, 673) but also bleeding-edge constructs that even mature tools struggle to handle.

Type checker correctness directly impacts software quality and developer productivity. When developers trust a type checker to validate their code, they expect accurate identification of genuine type errors without flagging correct code as erroneous. A type checker that produces excessive false positives leads to warning fatigue—developers begin ignoring alerts or adding unnecessary annotations to silence the tool, defeating the purpose of static analysis. Conversely, false negatives provide a dangerous illusion of safety: developers believe their code is type-safe when runtime TypeErrors lurk in production.

Figure 1 illustrates such a case:

```python
import itertools

def check(obj: object) -> None:
    match obj:
        case itertools.chain():
            print("Chain")
        case itertools.DoesNotExist():  # AttributeError at runtime!
            print("Unreachable")
        case _:
```

```
            print("Default")

if __name__ == "__main__":
    check(1)  # Crashes: 'module' object has no attribute 'DoesNotExist'
```

Figure 1: A false negative in mypy: pattern matching against a non-existent attribute passes type checking but crashes at runtime with an AttributeError.

The code in Figure 1 passes type checking in mypy without warnings, yet crashes at runtime with an AttributeError—the `itertools` module has no DoesNotExist attribute. This is a classic false negative: the type checker's static analysis failed to catch that the pattern match references a non-existent class, providing developers with false confidence in code correctness.

As the reliance on these tools in continuous integration pipelines grows, so does the importance of their correctness.

Yet these new tools remain largely unverified against real-world bug patterns. The Python Typing Council maintains an official conformance test suite [22] that validates type checker behavior against the Python typing specification. While established tools like pyrefly are included, newer implementations like ty (currently in beta) are not yet part of the conformance testing framework. When these tools are evaluated, conformance scores vary dramatically: ty achieves only 15% conformance, pyrefly 58%, and zuban 69% [27]. However, conformance scores measure adherence to the specification, not robustness against real-world bugs that fall outside the spec's scope. The current validation paradigm relies heavily on user-reported bugs, which creates a substantial time lag between bug introduction and detection. Oh and Oh found that type checker bugs persist for an average of 82 days, with 30% taking longer than a month to fix [17]. This reactive approach cannot keep pace with the rapid introduction of new features—Python 3.12 and 3.13 alone introduced PEP 695 (type parameter syntax) and PEP 742 (narrowing types), each creating new opportunities for type checker bugs to emerge. There is currently no systematic approach to proactively generating test cases that expose these bugs before users encounter them in production.

Existing automated testing approaches have well-documented limitations. Random fuzzing excels at generating syntactically valid code and detecting memory errors, but research has shown it struggles to trigger subtle semantic bugs that require specific preconditions or sequences of operations [29, 10]. Manually curated test suites, while high-quality, face inherent scalability challenges as system complexity grows and new features are added. What is needed is an approach that combines the coverage of automated generation with the semantic richness of real-world bug patterns.

Our key insight is that closed GitHub issues from type checker repositories represent ideal seeds for proactively generating novel test cases. Unlike random code, these issues encode confirmed, fixed bugs—real edge cases that developers already encountered and that tool maintainers deemed important enough to patch. By mining issues from python/mypy, facebook/pyrefly, astral-sh/ty, microsoft/pyright, and zubanls/zuban, we obtain a curated dataset of problematic patterns spanning protocols (PEP 544), TypedDict totality (PEP 589), ParamSpec decorators (PEP 612), type guards (PEP 647), and other advanced constructs. By using these historical patterns as seeds for automated mutation, we generate entirely new test cases that cause disagreements between checkers. Since disagreement implies at least one checker is incorrect, this approach exposes latent bugs in type checker implementations—enabling maintainers to fix edge cases proactively, before users encounter them in production.

Pytifex implements this "bug-seeded mutation" strategy by feeding 3–5 real bug examples to a large language model (Gemini), which generates novel code variations targeting similar typing patterns. The tool iterates until it collects the requested number of disagreement examples, meaning 100% of output examples are guaranteed disagreements—though reaching that count may require multiple generation loops depending on seed quality. In preliminary experiments, approximately 58% of generated examples per batch successfully triggered disagreements (11 of 19 in one batch). This efficiency aligns with prior work showing that bug-seeded approaches significantly outperform random generation: Patra & Pradel (FSE 2021) demonstrated that semantic bug seeding improved detection rates from 7% to 53% compared to artificial mutation [19]. The curated GitHub seeds provide the semantic guidance necessary to hit genuine type system edge cases.

Pytifex uses runtime testing as its primary oracle for establishing ground truth. When generated code crashes with a type-related exception, any checker that reported "OK" is definitively wrong—a false negative. This provides an objective, automated oracle that scales to thousands of examples without human judgment. Consensus-based approaches fail when all tools share the same blind spot; manual review is expensive and subjective. Runtime crashes admit no such ambiguity: the code either fails or it doesn't. However, this oracle is asymmetric: we can definitively prove false negatives (missed bugs), but cannot always prove false positives (spurious errors)—code may run successfully without triggering every potential bug path.

This research is motivated by the need for systematic, automated evaluation of the rapidly evolving Python type checker ecosystem. Unlike reactive approaches that wait for users to discover bugs, Pytifex proactively generates novel adversarial test cases from historical bug patterns, identifying correctness issues before they impact production codebases. This enables developers to choose reliable tools with confidence and provides type checker maintainers with concrete, actionable feedback to improve consistency across the ecosystem. As Python's type system continues to evolve and new tools proliferate, such proactive automated validation becomes not just useful but essential.

1.4 Proposed Solution: Pytifex

To address the validation gap in emerging Python type checkers, we present Pytifex, an automated differential testing agent that proactively generates test cases designed to expose disagreements between type checkers. Unlike conformance testing, which validates specification compliance, or reactive approaches that wait for user reports, Pytifex employs a bug-seeded mutation strategy: it mines closed issues from type checker repositories, extracts real-world bug patterns, and uses these as seeds to generate novel code variations via a large language model.

The key insight is that bugs tend to cluster around specific language features and edge cases. By learning from historical bugs—cases where type checkers previously disagreed or failed—Pytifex can systematically explore the space of potential disagreements. When the tool generates code that causes different type checkers to produce conflicting results (e.g., mypy reports an error while pyrefly accepts the code), this signals either a bug in one checker or an ambiguity in the specification.

Pytifex validates its findings using a runtime testing oracle: if generated code crashes with a TypeError, KeyError, or AttributeError at runtime, this definitively proves a false negative—at least one type checker should have caught the error statically but failed to do so. This oracle is asymmetric (we can prove false negatives but not always false positives),

but it provides concrete evidence of false negatives without requiring manual inspection.

We evaluate Pytifex on four type checkers spanning the maturity spectrum: the established mypy (v1.19.1), the recently released pyrefly (v0.50.1) and zuban (v0.4.2), and the alpha-stage ty (v0.0.1-alpha.32). Pytifex achieves a 58% per-batch success rate in generating disagreement-triggering code. Crucially, the agent iterates until the requested number of disagreements is collected, guaranteeing that 100% of output examples represent genuine type checker conflicts—efficiency we attribute to the semantic guidance of historical bug patterns.

1.5 Contributions

1. **Pytifex, the first automated differential testing agent for Python type checkers** that proactively generates test cases to expose implementation disagreements, rather than waiting for user-reported bugs.

2. **A bug-seeded mutation methodology** that mines historical bug reports from type checker repositories and uses LLM-guided generation to produce semantically meaningful test cases. We achieve a 58% per-batch success rate, with 100% of output examples guaranteed to represent genuine disagreements.

3. **A runtime oracle for ground truth** that definitively identifies false negatives without manual inspection—when generated code crashes with a type error, we prove which checker failed to catch it statically.

4. **An empirical evaluation demonstrating Pytifex's effectiveness** at exposing type checker disagreements and identifying false negatives through runtime validation. We characterize four major disagreement patterns across TypedDict handling, generic Self types, NewType validation, and TypeGuard soundness—areas where checkers frequently diverge.

5. **Open-source release of Pytifex and our generated test suite** at [https://github.com/benedekaibas/pytife demo], enabling reproduction and future research on type checker validation.

1.6 Ethical Implication

1.7 Paper Organization

The remainder of this paper is organized as follows. **Section 2** surveys related work in differential testing, type checker validation, and LLM-based test generation, positioning Pytifex within the broader landscape of static analyzer testing. **Section 3** describes our methodology in three parts: (1) mining closed GitHub issues from type checker repositories to extract real bug patterns, (2) using LLM-guided mutation to generate novel test cases that target similar edge cases, and (3) establishing ground truth for evaluating type checker correctness through a runtime oracle combining execution tracing, beartype enforcement, and Hypothesis property-based testing. **Section 4** presents our evaluation on four type checkers—mypy, pyrefly, zuban, and ty—analyzing disagreement patterns and runtime-proven false negatives across advanced typing features including TypedDict totality, generic Self types, NewType validation, and TypeGuard soundness. **Section 5** discusses limitations, including the oracle's asymmetry in proving false negatives versus false positives, and outlines directions for enhancing Pytifex's evaluation methodology and expanding bug detection coverage.

2 Related Work

2.1 Type Checker Validation Approaches

The Python community has developed several approaches to validate type checker correctness, each with distinct strengths and limitations.

Conformance Testing. The Python Typing Council maintains an official conformance test suite that validates type checker behavior against PEP specifications [22]. This suite tests whether implementations correctly handle prescribed typing features, such as generic types (PEP 484), protocols (PEP 544), and TypedDict (PEP 589). However, conformance testing has significant gaps. First, coverage is incomplete—newer type checkers like ty (currently in beta) are not yet officially included in the testing framework, though independent evaluation is possible. Second, when type checkers are evaluated against the conformance suite, scores vary dramatically: independent testing shows ty achieving 15% conformance, pyrefly 58%, and zuban 69% [27]. Third, and most critically, conformance tests measure adherence to specification, not robustness against real-world edge cases that fall outside the spec's explicit scope.

Manual Test Suites. Type checker projects maintain their own test suites, typically comprising thousands of hand-written examples. Mypy's test suite contains over 10,000 cases spanning a decade of development [12]. While these suites provide high-quality coverage of known patterns, they suffer from inherent scalability limitations. As Xu et al. observe in their study of Java type systems, manual test suites "can hardly keep up with rapid increase in size and complexity" of modern type systems [29]. Each new PEP introduces features that require new test cases, and the combinatorial explosion of interactions between features makes exhaustive manual coverage infeasible. Moreover, these suites tend to focus on canonical usage patterns rather than adversarial edge cases—the unusual combinations most likely to expose bugs.

Reactive Bug Detection. In practice, many type checker bugs are discovered through user reports. Developers encounter edge cases during development, file GitHub issues, and maintainers patch the bugs. This reactive approach creates substantial time lag between bug introduction and detection. Oh and Oh found that type checker bugs persist for an average of 82 days, with 30% requiring over a month to fix [17]. This delay is particularly problematic given Python's rapid evolution—Python 3.12 and 3.13 introduced major typing features (PEP 695, PEP 742) that created new opportunities for bugs before existing tools were fully validated.

Proactive Testing Techniques. Automated approaches like fuzzing and property-based testing have proven effective for compiler validation, but have seen limited application to Python type checkers. Tools like CSmith for C compilers demonstrate the potential of generative testing, yet Python's dynamic semantics and rapidly evolving type system present unique challenges that existing tools do not address.

The Validation Gap. Current approaches share a fundamental limitation: they are reactive rather than proactive. Conformance tests validate known specification requirements; manual suites encode previously discovered patterns; user reports identify bugs only after developers encounter them. There is no systematic approach to generating novel test cases that expose latent bugs before users encounter them in production. This gap motivates the need for automated, adversarial test generation that can proactively explore the space of potential type checker disagreements.

2.2 Differential Testing for Static Analyzers

Differential testing, formalized by McKeeman in 1998, is a methodology for detecting bugs by providing identical input to multiple implementations of the same specification and identifying discrepancies in their outputs [15]. Unlike traditional testing, which requires an oracle—a known correct answer for each test case—differential testing exploits the principle that consensus among independent implementations provides evidence of correctness. This approach is particularly valuable for domains like compilers and type checkers, where determining the correct output for arbitrary inputs is difficult and constructing test oracles is expensive [31]. When implementations disagree, at least one must be incorrect, enabling bug detection without manual verification.

Yang et al. demonstrated the effectiveness of differential testing for compiler validation with Csmith, a random C program generator that found over 325 previously unknown bugs in mature compilers including GCC and LLVM [31]. Their approach generated syntactically valid C programs, compiled them with multiple compilers, and compared outputs. Disagreements indicated compiler bugs, which were confirmed by examining the generated assembly or consulting language specifications. Critically, Yang et al. found that differential testing revealed bugs missed by extensive manual test suites, suggesting that consensus-based testing exposes edge cases that structured testing overlooks.

Le et al. extended differential testing with Equivalence Modulo Inputs (EMI), which generates program variants that preserve semantics under specific inputs [11]. By systematically deleting or modifying dead code, EMI creates programs that should produce identical results, exposing compiler bugs when outputs differ. While highly effective for compilers, EMI assumes executable semantics that can be compared at runtime—a property that type checker outputs lack.

Beyond compiler validation, differential testing has proven effective across diverse domains. Rigger and Su applied differential testing to database management systems, finding bugs in SQLite, MySQL, and PostgreSQL by generating semantically equivalent SQL queries and comparing results [24]. Kapus and Cadar used differential testing to validate symbolic execution engines, identifying disagreements between symbolic execution engines including KLEE and S2E [8]. These successes demonstrate that differential testing generalizes to any domain with multiple independent implementations of a common specification.

For type systems specifically, differential testing has seen limited but promising application. While the technique has been explored for other type systems, Python's gradual typing model and dynamic semantics present unique challenges that existing approaches do not address. PyTER [17] studied Python type checker bugs through issue mining rather than differential testing, finding that bugs persist for an average of 82 days—motivating the need for proactive detection techniques.

Python type checkers present an ideal target for differential testing. Multiple implementations (mypy, pyrefly, zuban, ty) claim to implement the same typing specification, yet they differ in maturity,implementation language, and optimization strategies. When these tools disagree on whether code is type-correct, the disagreement signals either a specification ambiguity or an implementation bug. However, a critical limitation emerges: unlike compilers where runtime execution provides ground truth, type checker disagreements alone cannot determine which tool is correct.

This motivates the need for an independent oracle. Unlike compiler testing where execution provides ground truth, type checker disagreements require external validation. We address this challenge by combining differential testing with runtime execution: when type

checkers disagree about whether code is correct, we execute the code to determine if runtime errors occur, providing an objective arbiter that neither manual analysis nor specification consultation requires.

2.3 Automated Test Generation

Automated test generation techniques have proven effective at discovering bugs in software systems, offering an alternative to manual test construction that can explore program behaviors systematically. These techniques broadly fall into three categories—mutation-based fuzzing, grammar-based fuzzing, and constraint-based approaches—with recent work exploring large language models as a complementary technique.

Mutation-based fuzzing generates test inputs by randomly modifying seed inputs, observing program behavior on the mutated inputs. Coverage-guided greybox fuzzing, exemplified by tools like American Fuzzy Lop (AFL), enhances this approach by using code coverage feedback to guide mutation toward unexplored program paths [1]. Böhme et al. model this process as a Markov chain, demonstrating that assigning energy inversely proportional to path frequency improves efficiency by gravitating toward low-frequency paths. Their approach found numerous bugs in widely-tested systems,showing that random mutation with coverage feedback can effectively explore complex program behaviors. However, for programs with highly structured inputs—such as programming languages purely random mutation often generates syntactically invalid inputs that are rejected before reaching interesting code paths.

Grammar-based fuzzing addresses this limitation by leveraging language grammars to generate syntactically valid inputs. Holler et al. introduced LangFuzz, which parses code fragments from existing test suites and mutates them according to language grammar rules [7]. Applied to JavaScript engines, LangFuzz discovered 105 severe vulnerabilities in Mozilla's interpreter within three months, demonstrating that grammar-aware generation can efficiently target language processors. This success highlight a key insight: domain-specific knowledge about input structure dramatically improves fuzzing effectiveness for language processors.

Constraint-based approaches, particularly symbolic execution, offer an alternative by systematically exploring program paths through constraint solving. Cadar et al. developed KLEE, a symbolic execution engine that achieved over 90% line coverage on GNU COREUTILS utilities and discovered bugs that had persisted for over 15 years [2]. KLEE symbolically executes programs, maintaining path constraints for each execution path and using constraint solvers to generate inputs that reach specific program locations. While powerful for achieving high coverage in systems code, symbolic execution faces scalability challenges: path explosion limits analysis to relatively small programs or short execution depths, and constraint solving becomes prohibitively expensive for complex path conditions.

Recent work has explored large language models for test generation. Unlike grammar-based approaches that capture only syntactic structure, LLMs can learn semantic patterns from training corpora, potentially generating more meaningful test cases. Deng et al. combined LLM generation with coverage-guided fuzzing in TitanFuzz, using the model to produce API call sequences that discovered bugs in deep learning libraries [4]. However, these approaches target general code generation or library APIs—not the specific challenge of exercising type system edge cases. Moreover, they generate inputs without domain-specific guidance about which language features are most likely to expose bugs.

For type checker testing, existing automated test generation techniques face specific

challenges. Random fuzzing, which generates inputs through random mutations, struggles with dynamic languages—Lukasczyk et al. found that lack of type information and Python's dynamic nature pose fundamental obstacles for automated test generation [14]. Grammar-based approaches can generate syntactically valid code but struggle to produce semantically meaningful programs. Padhye et al. note that while syntax generators can produce structurally valid inputs, ensuring semantic validity—inputs that satisfy domain-specific constraints—remains difficult [18]. For type checkers, this manifests as code that parses correctly but fails to exercise complex type system features like generic constraints, protocol inheritance, or type narrowing. Symbolic execution is poorly suited to type checkers, which perform static analysis rather than concrete execution, making symbolic execution's path-based reasoning less applicable. Moreover, none of these approaches leverage the rich signal available in existing bug reports—a source of information about precisely which language features and type constructs tend to trigger checker errors. This gap motivates our bug-seeded mutation approach, which combines domain knowledge from real bugs with LLM-based code generation to produce test cases that are both syntactically valid and semantically targeted at problematic type checker behaviors.

2.4 Bug-Seeded Mutation and MSR

Mining software repositories (MSR) extracts insights from development artifacts such as commit histories, bug reports, code reviews, and issue trackers to understand software evolution and failure patterns. Hassan and Xie survey the field, identifying how MSR techniques enable researchers to discover bug patterns, predict defects, and understand developer behavior from historical data [**?**]. For testing tools, MSR provides a crucial signal: real-world bugs reveal which language features, API combinations, and edge cases consistently challenge implementations.

Bug-seeded mutation leverages this insight by using known bugs as templates for generating new test cases. Patra and Pradel introduced SemSeed, which mines bug-fixing commits from open-source projects, extracts semantic bug patterns (e.g., incorrect API usage, missing null checks), and uses these patterns to seed mutations in new code [19]. Their approach achieved a 53% success rate in generating valid bugs, compared to just 7% for random fuzzing, demonstrating that real bug patterns provide strong priors for test generation. SemSeed's effectiveness stems from a key observation: bugs cluster around specific program constructs and API misuses, making historical bugs predictive of future failures. However, SemSeed operates at the level of runtime bugs (null pointer exceptions, API misuse) rather than type system bugs, and relies on pattern matching rather than semantic understanding of bug characteristics.

Several empirical studies demonstrate the value of mining type-related bugs specifically. Oh and Oh analyzed 300 Python type-related bugs from GitHub, characterizing bug persistence and fix patterns in Python type checking [17]. They identify common error patterns including incorrect type annotations, mismatched function signatures, and improper use of generic types. Chow et al. examined 2,766 type error fixes from open-source projects, categorizing errors by root cause and finding that certain constructs—particularly gradual typing boundaries and generic type parameters—account for disproportionate shares of bugs [3]. These studies reveal that type errors follow identifiable patterns, but neither work uses these patterns for automated test generation.

Our approach combines MSR with LLM-based mutation, mining closed issues from type checker repositories and using LLMs to generate semantically similar variations—addressing

the gap between pattern based mutation and the semantic understanding needed for type system edge cases. Unlike SemSeed's syntactic pattern matching, LLM-based mutation can generalize bug characteristics to new contexts while maintaining semantic validity. Recent work demonstrates the promise of this combination: Yang et al. developed WhiteFox, an LLM-guided compiler fuzzing approach that outperforms template based fuzzers by up to 8.2x in triggering optimizations [30]. While WhiteFox targets compiler optimizations rather than type checking, it demonstrates that LLM-guided mutation can exceed the effectiveness of pattern-based approaches—a principle we apply to the type checker domain.

2.5 LLM-Based Code Generation

Large language models have demonstrated significant capabilities in automated test generation, offering advantages over traditional approaches in terms of naturalness and diversity. Lemieux et al. developed CODAMOSA, which combines search-based software testing with LLM-guided generation [13]. Their approach conducts search-based testing until coverage improvements stall, then prompts an LLM to generate test cases for under-covered functions. On 486 benchmarks, CODAMOSA achieved statistically significantly higher coverage than pure search-based or LLM-only baselines, demonstrating that LLMs can effectively redirect search toward productive areas of the program space.

Schäfer et al. conducted a large-scale empirical evaluation of LLMs for unit test generation across 1,684 API functions in JavaScript packages [25]. Their TestPilot tool achieved median statement coverage of 70.2% and branch coverage of 52.8%, substantially outperforming the feedback-directed test generation technique Nessie (51.3% statement coverage, 25.6% branch coverage). Critically, 92.8% of generated tests showed less than 50% similarity with existing tests, indicating that LLMs produce novel test cases rather than memorizing training data. However, their evaluation also revealed challenges: generated tests sometimes fail to compile, require iterative repair through re-prompting, and struggle with complex API interactions requiring deep understanding of program semantics. While these results demonstrate LLM effectiveness for runtime test generation, their application to static type checker testing remains unexplored.

Beyond general test generation, Deng et al. demonstrated that historical bug-triggering code can guide LLM generation toward edge cases [**?**]. Their FuzzGPT approach found 76 bugs including 11 high-priority issues in deep learning libraries, suggesting that LLM semantic understanding combined with bug patterns can exceed template-based approaches. This finding motivates seeding LLM generation with real type checker bugs rather than relying on zero-shot prompting.

Existing approaches to type checker validation rely on conformance suites with incomplete coverage, manual test suites that cannot scale, or reactive bug reports that lag behind feature development. Automated techniques—fuzzing, symbolic execution, and LLM-based generation—have proven effective in adjacent domains but have not been combined for type checker testing. Pytifex addresses this gap by integrating differential testing across multiple type checkers, bug-seeded mutation from mined GitHub issues, LLM-based code generation, and runtime execution as an independent oracle.

3 Methods

3.1 Overview and System Architecture

Pytifex employs a four-stage pipeline to systematically generate test cases that expose type checker bugs: (1) mining bug reports from type checker repositories, (2) LLM-based mutation to generate semantically similar variations, (3) differential testing across multiple type checkers to identify disagreements, and (4) runtime validation to establish ground truth.

Figure 3.1 illustrates this architecture.

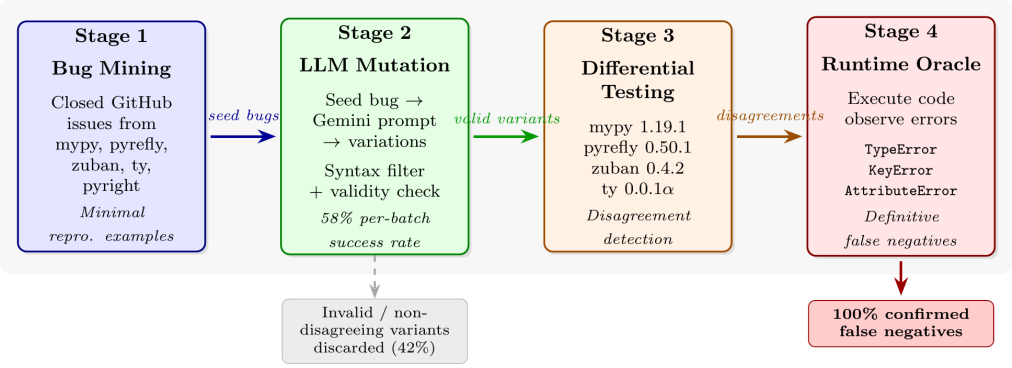

Figure 3.1: Pytifex system architecture. Seed bugs mined from closed GitHub issues are mutated by Gemini (Gemini 2.5 flash) to produce semantically similar variants that may trigger related failures across different type checkers. Valid variants are tested across four type checkers to identify disagreements. Disagreement-triggering code examples are executed at runtime; `TypeError`, `KeyError`, or `AttributeError` constitutes definitive evidence of a false negative. The pipeline guarantees that 100% of final outputs are confirmed false negatives.

The pipeline's design reflects two key insights from our analysis of related work. First, differential testing alone cannot determine which type checker is correct when disagreements occur, it can only signal that at least one implementation is wrong. Second, historical bug reports encode precisely which language features and type constructs tend to trigger checker failures, making them valuable seeds for generating new test cases.

3.2 Mining Type Checker Bug Reports

Pytifex mines seed bugs from five type checker repositories: mypy, pyrefly, zuban, ty, and pyright. Each pipeline run queries up to 100 issues per repository, sorted by most recent update, rather than scraping everything upfront. This recency-biased sampling means the seed pool shifts naturally as repositories evolve. Recently closed bugs enter the query window while older issues fall out. This allows Pytifex to incorporate emerging patterns without manual curation. The same issue may be mined across multiple runs, but the most recent bugs are consistently prioritized. Even though differential testing does not apply to pyright, the quality and variety of cases reported in its issue tracker make it worth including in the dataset.

3.2.1 Mining Process

Pytifex queries the GitHub REST API for closed issues labeled as bugs from each repository, sorted by most recent update. This retrieves up to 100 candidates per repository. We then apply post-fetch filtering to ensure seed quality:

Bug confirmation filtering. Issues closed as "not planned" are excluded since they represent `wontfix` decisions, `duplicates`, or reports that maintainers determined were not actual bugs. Only issues closed with resolutions indicating a genuine bug fix are retained.

Code extraction. From the remaining issues, we extract Python code blocks. An issue contributes a seed example if its body contains either a fenced code block (`python` or `py`) or, for Pyrefly, a sandbox URL encoding executable Python. We require at least 50 characters to exclude trivial snippets while permitting the minimal reproducible examples typical of well-filed bug reports.

Sampling. After shuffling the filtered issues, we extract up to 5 code examples per repository. A single issue may contain multiple code blocks that is why our limit applies to extracted examples, not issues. This multi-stage filtering reduces each repository's candidates to a handful of seeds, leaving approximately 10–25 total seed examples per pipeline run across all five repositories, with variation depending on repository maturity and the prevalence of extractable code in bug reports.

The recency bias serves an intentional purpose: recently closed bugs tend to exercise newer language features (e.g., PEP 695 type parameter syntax, PEP 742 TypeIs) that are more likely to expose disagreements in less mature type checkers. This seed based mutation strategy draws on FuzzGPT [Deng et al., 2024], which demonstrated that LLMs, when primed with code that has caused bugs in the past, generate significantly more programs that expose edge cases than zero-shot generation.

3.3 Illustrative Example: From Mined Seed to New Finding

To demonstrate the pipeline end-to-end, we trace one seed from mining through mutation to a confirmed type checker bug.

Code Listing 3.1(a) shows the original seed, mined from mypy issue #18524. The bug report described a false positive in mypy 1.14.1: mypy incorrectly marked match arms as unreachable when pattern matching on type objects with `--warn-unreachable` enabled. The code is valid Python that executes without error at runtime. Mypy's diagnostic was wrong. (This bug has since been fixed; the original false positive required the `--warn-unreachable` flag.) **Code Listing 3.1(a): Original seed from mypy #18524 (false positive).**

```python
import builtins
import types

def frobulate(field_type: type) -> str:
    match field_type:
        case builtins.int:
            ret = "foo"
        case builtins.str:    # mypy: "Statement is unreachable"
            ret = "foo"
        case builtins.bytes:
```

```
            ret = "foo"
        case builtins.bool:
            ret = "foo"
        case types.NoneType:
            ret = "foo"
        case _:
            return "bar"
    return ret
```

Given this seed, the LLM generated a variation that explores a different construction in the same feature area: matching on `type()` with positional sub-patterns, combined with generics and the `Self` type.

Code Listing 3.1(b): LLM-generated variation (adapted for presentation).

```
from typing import TypeVar, Self, Union, Any
import copy

T = TypeVar("T")

class Box[T]:
    def __init__(self, value: T) -> None:
        self.value = value

    def copy(self) -> Self:
        return type(self)(copy.deepcopy(self.value))

    def unwrap(self) -> T:
        return self.value

class IntBox(Box[int]):
    def double(self) -> Self:
        return type(self)(self.value * 2)

class StrBox(Box[str]):
    def upper(self) -> Self:
        return type(self)(self.value.upper())

def inspect_box_return(box_instance: Union[Box[Any], IntBox, StrBox]) -> str:
    copied_box = box_instance.copy()

    match type(copied_box):
        case type(IntBox(0)):   # TypeError at runtime
            return f"Copied an IntBox. Value: {copied_box.unwrap() * 3}"
        case type(StrBox("")):
            return f"Copied a StrBox. Value: {copied_box.unwrap().lower()}"
        case type(Box(None)):
            return f"Copied a generic Box. Value: {copied_box.unwrap()}"
        case _:
```

```
            return f"Unknown Box type: {type(copied_box).__name__}"

if __name__ == "__main__":
    print(inspect_box_return(IntBox(5)))   # Crashes here
```

Unlike the original seed, which was valid code that mypy incorrectly rejected, the variation contains a genuine bug: `case type(IntBox(0))` attempts to match on `type()` with a positional sub-pattern, but `type` does not define `__match_args__`. Python's runtime raises `TypeError: type() accepts 0 positional sub-patterns (1 given)` on every execution.

When tested across four type checkers, mypy, pyrefly, and zuban all correctly rejected the code, reporting that `type` does not define `__match_args__`. However, ty accepted the code, emitting only an unrelated warning about a possibly missing attribute. This constitutes a false negative: ty missed a bug that crashes deterministically at runtime. The pipeline's Phase 1 evaluation confirmed this verdict automatically with confidence 0.95.

This example illustrates a key property of seed-based mutation: the LLM did not reproduce the original false positive. Instead, it explored a related construction in the same feature space (match on type) that exposed a different scenario which in this case is a false negative in a different checker. The pipeline's value lies not in replicating known bugs but in using them as starting points to discover new ones.

3.4 LLM-Based Test Case Generation

Pytifex uses the mined seed examples to construct prompts that guide an LLM toward generating code likely to cause type checker disagreements. Rather than asking the LLM to generate test cases from scratch which tends to produce ordinary programs that all checkers handle identically [Deng et al., 2023], we prime it with real code examples that have caused false positives or false negatives in at least one type checker and explicit divergence targets.

3.4.1 Prompt Construction

Each generation prompt contains three components:

- 1 **Seed examples**. Up to 5 code examples from the mined seed pool, each annotated with its source repository, issue number, labels, and whether the original bug was a false positive or false negative. This metadata helps the LLM understand why each seed triggered a bug, not just what the code does.
- 2 **Divergence patterns**. A curated set of feature interaction patterns known to cause checker disagreements such as Protocol methods with default arguments (PEP 544), `TypedDict` with mixed Required/NotRequired inheritance (PEPs 589, 655), and `ParamSpec` applied to classmethods (PEP 612). These patterns serve as additional generation targets beyond what the seeds demonstrate.
- 3 **Mutation strategy**. Explicit instructions to mutate seeds into novel variations, not reproduce them: combine different patterns (e.g., `TypedDict` + `Protocol`), mutate seeds by changing types, adding generics, or wrapping in decorators, and probe the boundaries of what checkers catch. If a seed shows a false positive then the LLM is instructed to mutate it toward cases that other checkers also mishandle. If a seed shows a false negative the the LLM is instructed to explore the detection boundary.

Every generated example is required to reference a specific seed issue, establishing the source issue from the generated test case back to the real bug that inspired it. Examples without valid GitHub source issues are discarded.

3.4.2 Mutation-Filter Loop

The pipeline operates in a mutation-filter loop, iterating until a target number of disagreements is reached or a maximum number of attempts is exhausted. The outer loop is implemented in `generate_with_filtering` (Code Listing 3.3), which coordinates generation, testing, filtering, and refinement.

Code Listing 3.3: Outer mutation-filter loop (simplified from `pipeline.py`).

```
while len(collected) < target_count and attempt < max_attempts:
    attempt += 1
    # Step 1: Mutate - rotate through seeds for variety
    start_idx = (attempt - 1) * 3 % len(seed_examples)
    batch_seeds = seed_examples[start_idx:start_idx + 5]
    prompt = build_seed_based_prompt(batch_seeds, batch_size)
    response = agent.predict(prompt)
    parsed = generate_json.parse_generated_content(response)

    for item in parsed:
        example = Example(id=item["id"], code=item["code"],
                          metadata=item.get("metadata", ""),
                          seed_issue=extract_seed_issue(item.get("metadata", "")))
        # Step 2: Test - run all four type checkers
        example.results = run_all_checkers(example.code)
        # Step 3: Filter - keep only disagreements
        if has_disagreement(example.results):
            collected.append(example)
        else:
            # Step 4: Refine - try to create a disagreement
            refined = refine_example(agent, example, max_refinements)
            if refined:
                collected.append(refined)
```

Each iteration proceeds as follows:

- 1 **Mutate**. The LLM produces a batch of candidate programs (default: 15 per batch) by mutating the seed examples according to the prompt. Successive batches use a rotating window over the seed pool so that different batches draw on overlapping but distinct seed subsets, promoting diversity across iterations.
- 2 **Test**. Each candidate is written to a temporary file and executed sequentially by all four type checkers with default configurations and a 30-second timeout. The function `run_checker_on_code` (Code Listing 3.4) handles individual checker invocation.

Code Listing 3.4: Running a single type checker on generated code (from `pipeline.py`).

```python
def run_checker_on_code(code: str, checker_name: str,
                        command: list[str]) -> CheckerResult:
    temp_path = os.path.join(os.getcwd(), f"_pytifex_temp_{os.getpid()}.py")
    with open(temp_path, "w") as f:
        f.write(code)
    try:
        result = subprocess.run(command + [temp_path],
                                capture_output=True, text=True, timeout=30)
        output = result.stdout + result.stderr
        status = "ok" if result.returncode == 0 else "error"
        if "error" in output.lower() and "0 error" not in output.lower():
            status = "error"
        return CheckerResult(status=status, output=output.strip())
    except subprocess.TimeoutExpired:
        return CheckerResult(status="error", output="Timeout")
    finally:
        os.unlink(temp_path)
```

- 3 **Filter**. A candidate exhibits a disagreement if at least one checker's status differs from another's. The filtering logic (Code Listing 3.5) is intentionally minimal:

Code Listing 3.5: Disagreement detection (from `pipeline.py`).

```python
def has_disagreement(results: dict[str, CheckerResult]) -> bool:
    statuses = [r.status for r in results.values()]
    return len(set(statuses)) > 1
```

Formally, let S = {s , s , s , s } be the set of checker statuses, where each s {ok, error}. A disagreement exists when |S| > 1 that is, not all checkers agree. For example, if mypy, pyrefly, and zuban report error but ty reports ok, the candidate is a disagreement. Candidates where all four checkers agree are discarded.

- 4 **Regenerate**. Examples that pass through the Filter step without exhibiting a disagreement where all four checkers reported the same status undergo up to two regeneration attempts. The regeneration prompt includes the example's source code and the actual output from each checker, asking the LLM to minimally modify the code to induce a disagreement. Strategies include adding a subtle type error that only some checkers catch, fixing an obvious error while preserving a subtle edge case, or changing the typing pattern (e.g., introducing a `Protocol` or `TypeGuard`). Refinement is iterative: if the first attempt does not produce a disagreement, the second attempt regenerates the output of the first, giving the LLM two incremental adjustments rather than two independent attempts on the original. If neither attempt produces a disagreement, the example is discarded.

This closed loop design where checker feedback drives subsequent mutation distinguishes Pytifex from one-shot generation approaches. The refinement step recovers value from candidates that would otherwise be discarded: across our evaluation runs, approximately 71% of generated candidates exhibited disagreements with refinement contributing a portion of the successful examples.

3.5 Saving Code Examples

When the mutation filter loop completes, Pytifex persists all examples leading to disagreement to disk in a timestamped directory under `generated_examples/`. Each disagreement is saved as a standalone `.py` file containing the generated source code, and a companion `results.json` records the full pipeline run: the model used, the total number of candidates generated, the number of disagreements found, the per-batch success rate, and for each disagreement the raw output and status (`ok` or `error`) from every type checker. This structured output serves two purposes: it provides a reproducible artifact for subsequent evaluation and it preserves the provenance chain from mined seed to generated variant to checker outcome.

The results JSON also records the `seed_issue` field for each disagreement—the GitHub issue URL that inspired the generated code—enabling traceability from a confirmed finding back through the LLM mutation to the original bug report. Examples that lack a valid `seed_issue` reference are discarded during generation, ensuring that every saved disagreement has documented source.

3.5.1 Code Complexity Metrics

Beyond checker outputs, Pytifex computes and saves five structural metrics for each generated example, characterizing the complexity and type annotation density of the generated code. These metrics serve two purposes: they enable researchers to analyze whether disagreement rates correlate with code complexity, and they provide a quantitative profile of the test suite's diversity. The metrics are:

- **`loc`**: Lines of code excluding blank lines and comment-only lines.
- **`num_functions`**: Total number of function and async function definitions.
- **`type_imports`**: Count of imports from `typing`, `typing_extensions`, and `collections.abc`—a factor for the density of typing constructs.
- **`type_density`**: Computed as `(typed_objects - untyped_objects) / loc`, where typed objects include annotated parameters, return types, and annotated assignments, and untyped objects include parameters without annotations (excluding `self`/`cls`), functions without return annotations, and bare assignments. A positive value indicates a predominantly annotated codebase while a negative value indicates sparse annotations.
- **`internal_calls`**: Number of functions that are locally defined and call other locally defined functions to measure internal coupling.

Code Listing 3.6 shows an excerpt from a real `results.json` entry, illustrating how checker outputs, statuses, seed provenance, and code metrics are stored together for a single disagreement.

Code Listing 3.6: Excerpt from `results.json` for a generated disagreement.

```
{
  "filename": "typeddict-total-inheritance-divergence.py",
  "seed_issue": null,
  "metrics": {
    "loc": 62,
    "num_functions": 2,
```

```
    "type_imports": 8,
    "type_density": 0.1129,
    "internal_calls": 1
  },
  "statuses": { # NOTE: The statuses are simplified for the paper. Actual statuses include the fu
    "mypy": "error",
    "pyrefly": "ok",
    "zuban": "error",
    "ty": "error"
  }
}
```

In this example, `type_density` of 0.1129 indicates a moderately annotated file (62 lines, 8 typing imports), and the statuses show a 3-vs-1 disagreement: mypy, zuban, and ty report errors while pyrefly reports `ok`, signaling that pyrefly may have missed a genuine type error.

3.6 Establishing Ground Truth: The Evaluation Oracle

Differential testing identifies disagreements but cannot determine which checker is correct. A critical question remains: when mypy reports `ok` and ty reports `error`, which one is right? To answer this, Pytifex employs a multi-phased evaluation system that combines runtime execution, property-based testing, AST-level specification analysis, and static flow analysis to establish ground truth with varying degrees of confidence.

The evaluation system processes each disagreement through multiple independent oracles, each contributing evidence toward a final verdict per checker. The phases are not sequential filters since all phases run on every disagreement and their findings are aggregated by the `determine_verdicts` function (Section 3.X) that resolves conflicts by preferring higher-confidence evidence.

3.6.1 Phase 0: AST-Based PEP Oracle

Phase 0 performs source-level AST analysis to identify definitive violations of Python typing PEP specifications, independently of any type checker output. The oracle parses the generated source code and applies a list of rule based checks derived from PEPs 484, 526, 544, 586, 589, 591, 612, 634, 646, 647, 655, 673, 692, 695, 696, 698, 705, and 742. Each rule targets a specific construct. For example, `TDICT003` flags direct access to a `NotRequired` TypedDict field without a membership check (PEP 589), `FINAL003` detects overriding a `@final` method in a subclass (PEP 591), and `LSP001` identifies Liskov Substitution Principle violations in method overrides (PEP 484).

Each finding carries a confidence score, and only findings with confidence 0.85 are retained for verdict determination. The threshold exists because certain AST-based checks are inherently more ambiguous than others. For instance, detecting that a `@final` method is overridden in a subclass (rule `FINAL003`) can be determined with high certainty from the AST alone—the decorator and the method name in a subclass are unambiguous syntactic facts—so this rule emits confidence 0.95. In contrast, detecting whether a `TypedDict` field access violates `NotRequired` constraints (rule `TDICT003`) requires tracking field inheritance across multiple class definitions, which introduces opportunities for misresolution when aliases or re-exports are involved; such rules emit confidence 0.90. The 0.85 threshold

filters out lower-confidence heuristic checks that risk producing false verdicts while retaining rules grounded in clear PEP requirements.

The oracle then evaluates each type checker by parsing its raw output into structured diagnostics (line number, error code, message, severity) and matching them against the oracle's findings. A finding is considered matched if the checker reported an error within 5 lines of the violation and the error code or message keywords correspond to the violation category. The verdict logic is:

- **CORRECT**: All oracle findings matched by checker diagnostics.
- **INCORRECT**: One or more findings not reported by the checker (false negative).
- **UNCERTAIN**: No oracle findings exist, or upstream checker errors may have blocked detection of the violation.

Code Listing 3.7 shows a generated example that Phase 0 identified as containing a TypedDict totality violation. The `FullServiceConfig` class inherits from both `BaseConfig` (total=True) and `OptionalFeatures` (total=False). When `FullServiceConfig` sets `total=True`, fields inherited from the `total=False` parent become Required per PEP 589. The dict literal `cfg1` omits `debug_mode` and `log_level`, which the oracle flags as missing required keys. Three of four checkers (mypy, zuban, ty) correctly reported the error; pyrefly did not, constituting a false negative.

Code Listing 3.7: Generated example triggering a Phase 0 TypedDict totality finding.

```
class BaseConfig(TypedDict):           # total=True by default
    host: str
    port: int

class OptionalFeatures(TypedDict, total=False):
    debug_mode: bool                   # implicitly NotRequired
    log_level: Literal["INFO", "WARN", "ERROR"]

class FullServiceConfig(BaseConfig, OptionalFeatures, total=True):
    api_key: str                       # Required (total=True)
    timeout_seconds: NotRequired[int]  # explicitly NotRequired

# Phase 0 oracle finding: cfg1 is missing required keys
# 'debug_mode' and 'log_level' (promoted to Required by total=True)
cfg1: FullServiceConfig = {
    "host": "localhost", "port": 8080,
    "api_key": "secret-key-123",
    "timeout_seconds": 60,
    # debug_mode and log_level omitted - PEP 589 violation
}
```

The oracle's strength lies in its independence: it derives ground truth from PEP specifications rather than from checker consensus, avoiding the circularity of using checkers to validate themselves. However, it is limited to constructs with unambiguous specification-level rules and cannot evaluate advanced features like `TypeGuard`, `ParamSpec`, or `TypeVarTuple`, which require semantic reasoning beyond AST pattern matching.

3.6.2 Phase 1: Runtime Crash Detection

Phase 1 executes the generated code and catches type-related runtime exceptions—`TypeError`, `KeyError`, and `AttributeError`. A runtime crash constitutes the highest-confidence evidence of a type bug: if code raises `TypeError` at runtime, any type checker that reported `ok` is definitively incorrect (a false negative). This oracle is asymmetric—it can prove false negatives but cannot prove false positives—because code may run successfully without exercising every type-incorrect path.

The implementation goes beyond naive execution in three ways. First, it walks the full traceback to identify the root-cause line, not just the final frame, attributing the bug to the line in the generated source where the type violation originates rather than to library internals. Second, it inspects exception chains via `__cause__` and `__context__`, surfacing chained type errors that would otherwise be masked. Third, it isolates `try/except` bodies by extracting them via AST analysis and re-executing them independently. This surfaces type errors that the original code deliberately swallows—a common pattern in generated code where `try/except Exception` blocks mask `TypeError` or `KeyError` exceptions that should propagate. Bugs discovered through isolation receive a slightly reduced confidence of 0.95 (versus 1.0 for direct crashes) to reflect the contextual difference.

The illustrative example from Section 3.3 demonstrates Phase 1 in action: Code Listing 3.1(b) crashes with `TypeError: type() accepts 0 positional sub-patterns (1 given)` at line 108 when executed. Phase 1 catches this exception, attributes it to the `case type(IntBox(0))` pattern, and marks ty as `INCORRECT` because ty reported `ok` while the code crashes deterministically.

3.6.3 Phase 2: Hypothesis Property-Based Testing

Phase 2 employs Hypothesis, a property-based testing framework, to exercise code paths that Phase 1's single execution may not reach. Rather than relying on the `if __name__ == "__main__"` block alone, Phase 2 systematically generates inputs for every user-defined function and class method in the generated code.

The process operates in five steps:

- 1 **Definition extraction**. The AST is parsed to identify all user-defined functions, constructors, and methods. Built-in names, dunder methods, and private functions are excluded to focus on the code's public API.
- 2 **Namespace construction**. The source code is executed once with `__name__` set to a non-`"__main__"` value, building a live namespace containing all defined classes and functions. This step is necessary because Hypothesis requires callable objects, not AST nodes.
- 3 **Signature introspection**. For each callable, Python's `inspect.signature` and `typing.get_type_hints` extract parameter types and return annotations. These concrete type hints are mapped to Hypothesis strategies—`int` maps to `st.integers()`, `str` to `st.text()`, `list[int]` to `st.lists(st.integers())`, and so on. Parameters with `ParamSpec` components or unresolvable types are skipped rather than generating arbitrary values.
- 4 **Property test execution**. Each callable is tested with `@given` decorators using the generated strategies. The property under test is that calling the function with type-

conforming inputs should not raise `TypeError`, `KeyError`, or `AttributeError`. Hypothesis runs up to 30 examples per callable with no deadline, allowing slow-executing generated code to complete.

- 5 **Return type validation**. When a function declares a return type and the `typeguard` library is available, return values are validated against their declared type annotation, catching cases where a function's implementation silently returns an incompatible type.

Phase 2 also includes targeted tests that exercise specific call sites found in the source code's `if __name__ == "__main__"` block and class method chains, using AST analysis to construct invocation sequences that mirror the code's intended usage patterns.

To illustrate Phase 2's value, consider the generated file `loop-newtype-divergence.py` from our evaluation. Phase 1 found no crashes because the `if __name__ == "__main__"` block executed without error. However, Phase 2 discovered five `ValueError` bugs at line 6 by generating inputs such as empty tuples, single-element tuples, and strings—all type-conforming according to the declared `tuple[UserId | None, ItemId | None]` annotation but triggering unpacking failures. These bugs proved that all four checkers missed real type safety issues, as none flagged the vulnerable unpacking pattern. The evaluation JSON recorded these as:

```
"tier2_bugs": [
    {"line": 6, "type": "ValueError",
     "msg": "empty_tuple: not enough values to unpack (expected 2, got 0)"},
    {"line": 6, "type": "ValueError",
     "msg": "single_element_tuple: not enough values to unpack (expected 2, got 1)"}
]
```

Bugs discovered by Hypothesis receive confidence 0.85, reflecting that they are genuine runtime failures triggered by valid inputs, though the inputs are synthetically generated rather than reflecting the code's original execution path.

3.6.4 Phase 3: PEP Specification Compliance

Phase 3 applies a curated set of PEP-derived rules to each checker's output, determining whether the checker's behavior aligns with the official Python typing specifications. Unlike Phase 0, which analyzes the source code independently, Phase 3 examines the checker outputs themselves—matching their error messages against regex patterns derived from specific PEP requirements.

The rule set covers 24 PEPs and encompasses patterns including method override compatibility (PEP 484), protocol instantiation (PEP 544), `Final` reassignment and subclassing (PEP 591), `TypedDict` key requirements (PEPs 589, 655), `ParamSpec` and `Concatenate` usage (PEP 612), `TypeGuard` and `TypeIs` semantics (PEPs 647, 742), and `ReadOnly` TypedDict fields (PEP 705). Each rule is defined as a `PEPRule` containing a PEP number, a regex pattern, a human-readable description, and the expected correct behavior (`error` or `ok`). For example:

```
PEPRule(
    pep_number=591,
    pattern=r"[Cc]annot (?:assign|override|overwrite).*Final|Final.*reassign",
    rule_description="Final variables cannot be reassigned (PEP 591)",
    correct_behavior="error",
)
```

For each rule, the evaluator checks whether the checker's output contains a matching error message and whether the checker's overall status (error or ok) agrees with the PEP's prescribed behavior. If PEP 591 specifies that reassigning a `Final` variable should produce an error, and mypy reports such an error while ty does not, mypy is marked correct and ty incorrect for that rule. Phase 3 also incorporates a source-aware analysis pass that uses the AST oracle findings from Phase 0 and checks whether each checker reported errors near the identified violation lines, providing an independent cross-check.

Module import errors and `reveal_type` diagnostics are filtered out to avoid false matches—these are tool-specific behaviors unrelated to type correctness.

3.6.5 Phase 4: Static Flow Analysis and Design Differences

When Phases 0–3 leave a checker's verdict uncertain, Phase 4 performs additional static analysis covering constructs that earlier phases cannot evaluate. This includes import availability checks (whether a typing construct like `TypeIs` or `ReadOnly` is available in the target Python version or requires `typing_extensions`), variance constraint analysis, type narrowing flow through `TypeIs`/`TypeGuard`/`isinstance`/`match` statements, nominal type boundary enforcement for `NewType`, and match exhaustiveness verification.

Cases that remain uncertain after all four phases represent genuine design differences between type checkers—areas where the Python typing specification is ambiguous or where checkers make legitimately different approximation choices. These are documented as `UNCERTAIN` rather than forced into a potentially incorrect verdict.

3.6.6 Verdict Determination

The final verdict for each checker on each disagreement is determined by the `determine_verdicts` function (Code Listing 3.8), which aggregates evidence from all phases using a strict priority order. The function iterates over each checker and short-circuits at the first phase that produces definitive evidence.

Code Listing 3.8: Priority-based verdict determination (simplified from `comprehensive_eval.py`).

```
def determine_verdicts(tier1_bugs, tier2_bugs, tier3_findings,
                       checker_outputs, source_code,
                       tier4_findings=None, oracle_verdicts=None):
    verdicts = {}
    function_spans = extract_function_spans(source_code)
    tier1_bugs_only = [b for b in tier1_bugs
                       if b.confidence >= 0.85 and b.source == "tier1_runtime"]
    tier2_bugs_high = [b for b in tier2_bugs if b.confidence >= 0.85]
```

```
    for checker, output in checker_outputs.items():
        checker_error_lines = extract_checker_error_lines(output)

        # Phase 1: runtime crashes outrank everything
        if tier1_bugs_only:
            caught, missed = _check_bugs_against_checker(
                tier1_bugs_only, checker_error_lines, function_spans)
            if caught and not missed:
                verdicts[checker] = {"verdict": "CORRECT", "confidence": 0.9, "tier": 1}
                continue
            elif missed:
                verdicts[checker] = {"verdict": "INCORRECT", "confidence": 0.95, "tier": 1}
                continue

        # Phase 0: AST oracle
        ov = oracle_verdicts.get(checker)
        if ov and ov.verdict != "UNCERTAIN":
            verdicts[checker] = {"verdict": ov.verdict,
                                 "confidence": ov.confidence, "tier": 0}
            continue

        # Phase 2: Hypothesis property-based testing
        if tier2_bugs_high:
            caught, missed = _check_bugs_against_checker(
                tier2_bugs_high, checker_error_lines, function_spans)
            if caught and not missed:
                verdicts[checker] = {"verdict": "CORRECT", "confidence": 0.85, "tier": 2}
                continue
            elif missed and not caught:
                verdicts[checker] = {"verdict": "INCORRECT", "confidence": 0.85, "tier": 2}
                continue

        # Phase 3: PEP specification compliance
        # ... (pattern matching against PEP rules)
        # Phase 4: Static flow analysis
        # ... (import availability, variance, narrowing)

        # Fallback: no definitive evidence
        verdicts[checker] = {"verdict": "UNCERTAIN", "confidence": 0.5, "tier": 4}
    return verdicts
```

The priority order reflects confidence levels: Phase 1 runtime crashes (confidence 0.95) constitute irrefutable proof and therefore outrank all other evidence. Phase 0 AST oracle findings follow, as they are derived from unambiguous PEP specifications. Phase 2 Hypothesis-proven bugs (confidence 0.85) rank next, as they represent genuine runtime failures under valid inputs. Phase 3 PEP compliance matching (confidence 0.80) and Phase 4 static analysis complete the cascade.

A final false-positive detection step covers a specific scenario: if no phase found any vio-

lations and the oracle confirmed the source is violation-free, but a checker still reports errors, that checker is marked `INCORRECT` for producing false positives—provided the source does not use typing constructs beyond the oracle's coverage (e.g., `ParamSpec`, `TypeVarTuple`). If no phase provides definitive evidence, the verdict is `UNCERTAIN` with confidence 0.5.

3.6.7 Agent-Based Resolution of Uncertain Cases

For disagreements where all phases yield `UNCERTAIN`, Pytifex optionally invokes the Gemini LLM as a final arbiter. The agent receives the source code, the target checker's output, and all other checkers' outputs, and is asked to determine whether the checker's behavior is `CORRECT`, `INCORRECT`, or `UNCERTAIN`, citing the specific PEP section that supports its verdict. The agent's response is parsed for a structured JSON verdict; if parsing fails, the system falls back to extracting the verdict from prose. Agent-resolved verdicts are reported separately from phase-based verdicts, maintaining a clear distinction between deterministic evaluation and LLM-based judgment.

3.7 Implementation

Pytifex is implemented in Python 3.12 and uses the Google Gemini API (via `httpx`) for LLM-based code generation and agent-based evaluation. The pipeline is orchestrated by a command-line interface that supports four modes: `full` (generation + evaluation), `generate` (generation only), `check` (run type checkers on existing examples), and `eval` (evaluate existing results). Type checkers are invoked via `subprocess` with a 30-second timeout per file, using their default configurations. The tool depends on `hypothesis` and `typeguard` for Phase 2 property-based testing, and on `beartype` for supplementary runtime type enforcement.

All type checker outputs are real—they come from actual tool execution on the local system, not LLM simulation. The LLM is used exclusively for two purposes: generating candidate code from seed examples, and resolving uncertain verdicts in the agent-based fallback. The evaluation phases (0–4) are entirely deterministic and reproducible.

The tool is released as open source at [https://github.com/benedekaibas/pytifex-demo], enabling reproduction of our results and future research on type checker validation.

References

[1] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1032–1043.

[2] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 209–224.

[3] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the 46th International Conference on Software Engineering (ICSE '24)*. ACM. `https://doi.org/10.1145/3597503.3639184`

[4] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 423–435.

[5] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 209–220. `https://doi.org/10.1145/3540250.3549114`

[6] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Computer Aided Verification (CAV 2018)*. 12–19. `https://doi.org/10.1007/978-3-319-96142-2_2`

[7] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*. 445–458.

[8] Timotej Kapus and Cristian Cadar. 2019. Differential testing of symbolic execution tools. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1206–1209.

[9] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. 2022. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* 48, 8 (2022), 3145–3158. `https://doi.org/10.1109/TSE.2021.3079205`

[10] Seulbae Kim, Major Fayyaz, Junqing Shen, Sangheon Yun, Youngjin Kim, Camilo Xu, Taehoon Lee, Jaehan Na, Sihyun Kim, Manos Kapritsos, Heejo Bang, Sooel Kang, and Jun Hee Kim. 2024. RoboFuzz: Fuzzing Robotic Systems over Robot Operating System (ROS) for Finding Correctness Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. ACM, 1411–1423. `https://doi.org/10.1145/3650212.3680384`

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 216–226.

[12] Jukka Lehtosalo and Mypy Contributors. 2024. Mypy Test Suite. `https://github.com/python/mypy/tree/master/test-data` Accessed: 2025-01-15.

[13] 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 *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. 919–931.

[14] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated unit test generation for Python. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 9–24.

[15] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[16] Zeina Migeed and Jens Palsberg. 2020. What Is Decidable about Gradual Types? *Proc. ACM Program. Lang.* 4, POPL, Article 29 (2020), 29 pages. `https://doi.org/10.1145/3371097`

[17] Wonho 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 (ESEC/FSE '22)*. Association for Computing Machinery, New York, NY, USA, 922–934. `https://doi.org/10.1145/3540250.3549130`

[18] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 329–340.

[19] Jibesh Patra and Michael Pradel. 2021. Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE '21)*. ACM, New York, NY, USA, 906–918. `https://doi.org/10.1145/3468264.3468623`

[20] 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 (ICSE '22)*. 2019–2030. `https://doi.org/10.1145/3510003.3510038`

[21] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '21)*. 24–35. `https://doi.org/10.1109/SANER50967.2021.00012`

[22] Python Typing Council. 2024. Python Typing Conformance Test Suite. `https://github.com/python/typing/tree/main/conformance`. Accessed: 2025-01-30.

[23] Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-Time Static Type Checking for Dynamic Languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 462–476. `https://doi.org/10.1145/2908080.2908095`

[24] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. In *Proceedings of the ACM on Programming Languages (OOPSLA)*. `https://doi.org/10.1145/3428279`

[25] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105.

[26] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[27] Sinon. 2025. How Well Do New Python Type Checkers Conform? A Deep Dive into Ty, Pyrefly, and Zuban. https://sinon.github.io/future-python-type-checkers/. Accessed: 2025-01-30.

[28] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Implementation of Gradual Typing. In *Proceedings of the 12th Symposium on Dynamic Languages*. ACM, 121–132. https://doi.org/10.1145/2661105.2661111

[29] Wen Xu, Hyungon Moon, Sanidhya Kashyap, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, 147–161. https://doi.org/10.1145/3341301.3359662

[30] Chenyuan Yang, Yinlin Liu, Yuxuan Cao, Yuxiang Chen, Yisen Pang, Xiaoli Huang, Zhiyuan Zhang, Zhenkai Xu, Yuqun Zhang, Sarena Raharjo, Chengyu Pang, Yingwei Ma, and Lei Ma. 2024. WhiteFox: White-box compiler fuzzing empowered by large language models. In *Proceedings of the ACM on Programming Languages, Vol. 8, No. OOPSLA2*. 296.

[31] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 283–294. https://doi.org/10.1145/1993498.1993532