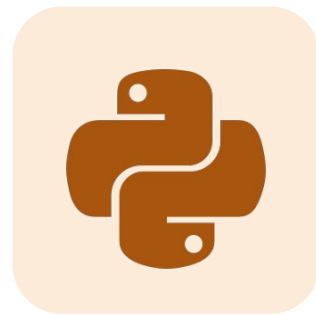


PYTIFEX · A MUTATION-BASED DIFFERENTIAL TESTING TOOL FOR PYTHON TYPE CHECKERS

Don't Blame the Type Checker:

Automatically Finding Who Is Right When Python Type Checkers Disagree



PYOHIO 2026

Benedek Kaibas

Gregory M. Kapfhammer

Department of Computer and Information Science

Allegheny College

<https://github.com/benedekaibas/pytifex>

Who is the creator of Pytifex?

Name: Benedek Kaibas

Affiliation: Currently entering Masters Studies in Computer Science

Currently working on the Clang Static Analyzer as a Google Summer of Code student @ LLVM

Interests:

- Static Analysis
- Compilers
- Functional programming
- Research and engineering

In case you want to talk with me, feel free to contact me at kaibas01@allegheny.edu. I am glad to chat on Discord!



Type annotations and type checkers for Python

An **annotation declares** what a value should be! A **type checker verifies it without running the code!**



```
def greet(name: str) -> str:  
    return "Hello, " + name
```



Type annotations: `str` and `-> str` declare the expected types for the `greet` function. Annotations are possible for a wide range of data types including lists, tuples, and dicts



The type checker

reads the annotations and flags mismatches to the Python programmer



`greet("Ada")`
types match — no error



`greet(42)`
int is not str — `TypeError`

The checker catches type mistakes **before the code ever runs** — no tests, no code execution. This is awesome, right? However, programmers need to invest time to add the annotations And there is the possibility that type checkers may make mistakes. **Let's explore further!**



Do Python codebases need type annotations?

Why would annotating your project **help you as a Python programmer?**

Open-Source Python Projects

- FastAPI
- Pydantic
- Django
- Flask
- ... and many others!

Other Open-Source Projects

- LLVM/Clang
- Swift
- Chromium
- Firefox
- ... and many others!

Type annotations help you, your IDE, your tools, and your AI agents!

So, what are the inputs and outputs of a Python type checker?



Four verdicts a type checker can surface

	checker says “OK”	checker says “ERROR”
code is FINE	 TRUE NEGATIVE correct — stays quiet	 FALSE POSITIVE false alarm on fine code ← a checker is wrong
code is BUGGY	 FALSE NEGATIVE misses the bug (prior slide) ← a checker is wrong	 TRUE POSITIVE correct — catches it

The two off-diagonal cells are the **key ways in which a checker is wrong!**



Exciting times for Python type checkers!



Mypy

Well-known type checker that is implemented in Python and may exhibit performance issues



Ty

New Rust-based type checker and language server from the teams at Astral and OpenAI



Pyrefly

New Rust-based type checker and language server from the team at Meta Open Source



Zuban

New Rust-based type checker and language server from David Halter of Jedi fame

Many developers use multiple type checkers in their Python project



Insights from the Python community



Are you really expected to run five type-checkers now?

TL;DR: Prioritize running as many type-checkers as possible on your test suite. Run at least one on your source code. <https://pyrefly.org/blog/too-many-type-checkers/>



Who or what should use a Python type checker now?

TL;DR: Of course, run a type checker in your CLI and your IDE. But, AI agents should run type checkers as well! <https://pyrefly.org/blog/type-checking-agentic-workflows/>

Blog post about the Python type checker conformance suite:
<https://pyrefly.org/blog/typing-conformance-comparison/>



Key challenge: type checker disagreement



```
class StringTransformer(Protocol):
    def transform(self, text: str) -> str: ...

class UpperCaser:
    # Bug: takes list[str] instead of str
    def transform(self, items: list[str]) ->
list[str]:
    return [item.upper() for item in items]

def apply_and_trim(t: StringTransformer, text:
str) -> str:
    result = t.transform(text)
    return result.strip()    # crashes: list has no
.strip()

uc = UpperCaser()
print(apply_and_trim(uc, "hello")) # Runtime
Error
```



mypy Code is fine!



pyrefly Code is fine!



ty Code is not fine!



zuban Code is not fine!

Who is right?



Why type checker bugs matters to you

Even if you are **not** a type checker developer, **Pytifex** is important to you!



Developers

Help to eliminate defects in type checkers, improving your experience with these tools



Researchers

Advance work in automated software testing techniques that can find bugs in real tools



Educators

Automatically generate many examples of subtle programs with type checking errors



Students

Get involved in research and development involving theory, practice, and LLMs



Why generate new Python code at all?

You can't wait for these bugs to appear — **you can instead proactively create them!**

ONE hand-written example

```
x: int = f()
reveal_type(x)
```

mypy ✓

pyrefly ✓

zuban ✓

ty ✓

They agree here — which tells you **nothing** about the next program. Wouldn't it be great if a tool could automatically generate the disagreements?



MANY novel Python programs

✓ agree

✓ agree

⚡ differ

✓ agree

⚡ differ

✓ agree

A program where checkers **disagree** is a candidate bug — one of them may be provably wrong. If one checker does not flag a type error and the others do, and you can find a way to make that program crash with a type error, then you found a real problem!

Automatically synthesize Python code that the type checkers have never seen, looking for the **disagreements** that expose a **false positive** or **false negative**. We built **Pytifex** to use GitHub **issue trackers** and **LLMs** to complete this task for you!



Automatically pressure testing type checkers

Generate unseen code · run every checker · keep the disagreements



1. Generate

Synthesize novel Python programs the checkers have never seen.



```
def f(x):  
    # function's body
```



2. Run all four checkers

Execute every checker on the same program — four independent verdicts.

mypy

pyrefly

zuban

ty



3. Compare

If the verdicts differ, at least one checker is wrong — a candidate bug in the type checker itself

Submit bug reports to GitHub
for relevant type checkers

Each disagreement is then judged **objectively** with the crash tests and property tests of Phase 1 & 2



How to evaluate the Python type checkers?

Pytifex follows **three principles** to ensure that it produces the best possible insights

1

STAY OBJECTIVE

Judge only on objective facts — every checker formats its report differently and leans on subjective heuristics

Pytifex must reason objectively

Do not pattern match on type checker output

Type checkers emit reports in differing formats

Type checkers often use varying heuristics

2

THE LLM CANNOT JUDGE

An LLM can hallucinate a verdict, so it can never be the arbiter of correctness or a checker disagreement

Focus on automatically generating Python programs for which there is convincing evidence of a type checker bug

3

TRUST OVER VOLUME

A false verdict is worse than none — keep only the results you can fully trust based on type checker output

Don't overwhelm the maintainers of type checkers with too many superfluous bug reports; find critical defects



Mine bug reports and generate new code

Generate unseen code · run every checker · keep the disagreements

1

SEED MINING

Mine GitHub Bug Reports

Fetch closed, confirmed-bug issues from upstream type-checker repos. Extract embedded Python snippets as seeds. Use these seeds as input to the code generation phase since they represent prior real-world defects

python/mypy

facebook/pyrefly

astral-sh/ty

zubanls/zuban

microsoft/pyright



2

CODE GENERATION

Generate Novel Variations with an LLM

A batch of seed snippets is sent to the LLM with a prompt to generate new code that targets similar edge cases. Create a batch of 15 examples per attempt, repeatedly working to create novel disagreements between checkers

Pytifex offers **automated** and **proactive** detection of potential limitations of Python type checkers



Mine bug reports and generate new code

Generate unseen code · **run every checker** · keep the disagreements

3

TYPE CHECKER EXECUTION

Run All Four Checkers

Every generated example is executed through all type checkers in a subprocess — real outputs, no simulation.
The goal is to have an automated process that can quickly produce disagreements among type checkers.



DECISION Do the checkers disagree?

YES

OUTPUT

Novel Disagreement Saved

Kept for evaluation. Stored in the generated examples with full type checker outputs to aid bug reporting.

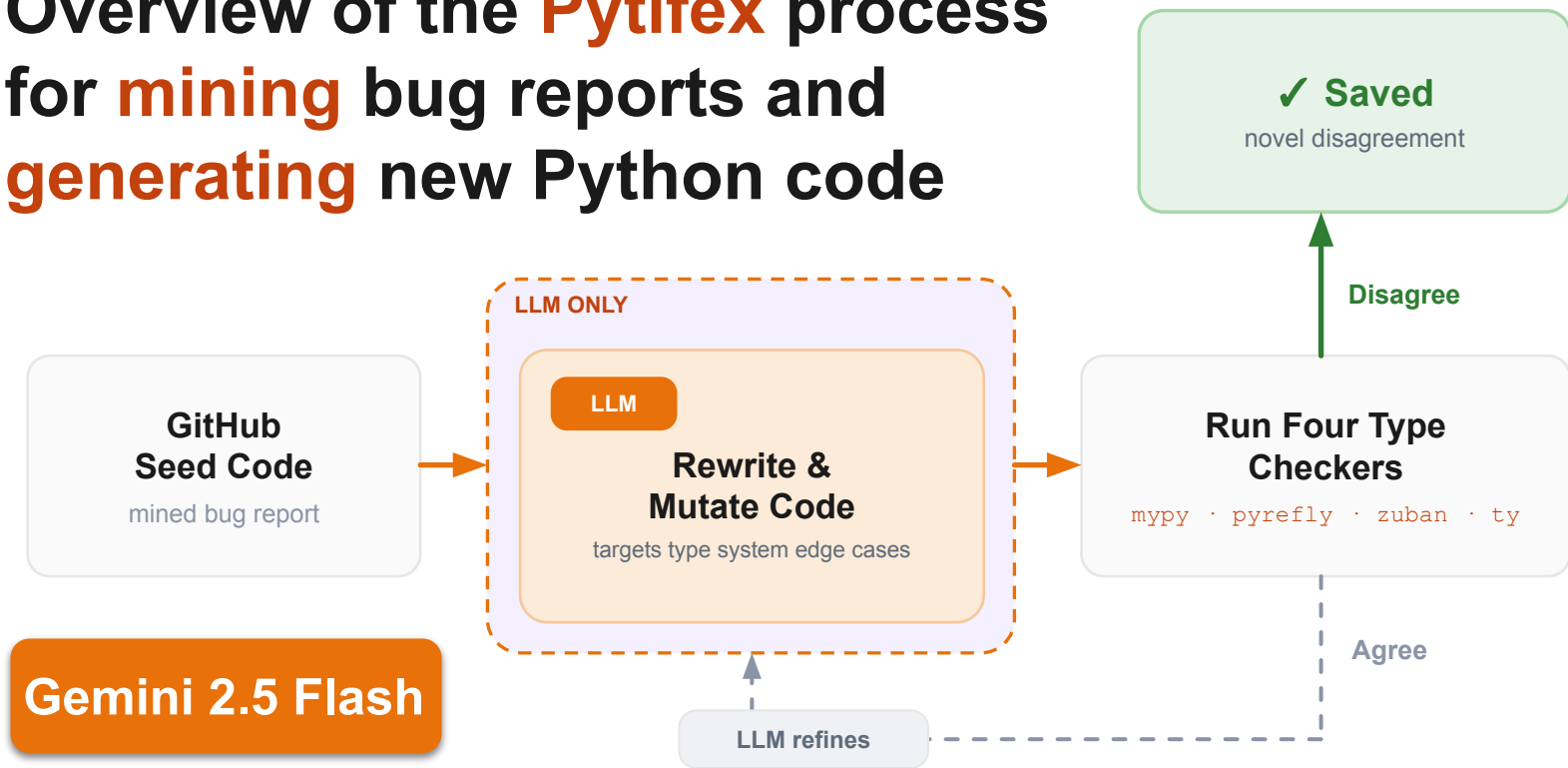
NO

REFINEMENT

LLM Refinement Pass

Send the type checker's feedback back to the LLM as a way to steer the search to a possible disagreement.

Overview of the **Pytifex** process for **mining** bug reports and **generating** new Python code



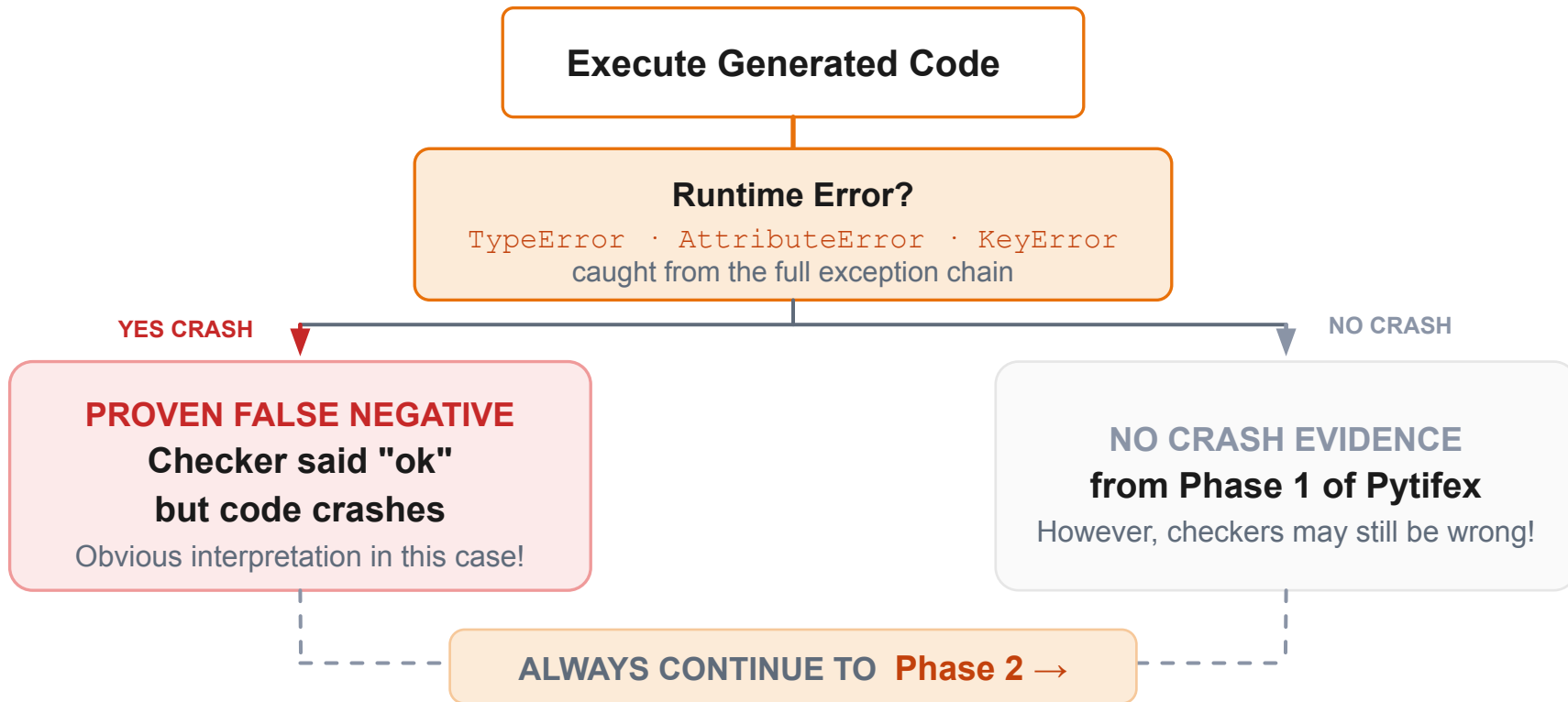
■ LLM involved — mutation & refinement ■ Checkers disagree — example saved

■ Checkers agree — LLM refines and retries



Reviewing Phase 1 of Pytifex

If there is a runtime crash, then there is objective proof when there is type checker disagreement





Enforcing type annotations with Beartype

In **Phase 2** of Pytifex the type annotations become **enforced runtime contracts**

```
from beartype import beartype

@beartype
def repeat(text: str, n: int) -> str:
    return text * n

repeat("hi", 3)      # ✓ "hihihi"
repeat("hi", "3")    # ✗ runtime type error

# by default, type annotations do not
# crash a Python program at runtime!

# beartype introduces enforced runtime contracts
```

WHAT IT DOES

Beartype rewrites each annotation into a **runtime check**. Any argument — or the return value — that breaks the declared type raises on the spot.

WHY PHASE II NEEDS IT

It proves every generated input is genuinely **type-conformant**, so a crash is a real type bug — and the return type is validated too.



Use Hypothesis to generate valid inputs

Phase 2 · hundreds of inputs from the type annotation, not the limited inputs generated by the LLM



```
from hypothesis import given
from hypothesis import strategies as st

@given(
    text=st.text(),
    n=st.integers(min_value=0, max_value=100),
)
def test_repeat(text, n):
    result = repeat(text, n)
    assert len(result) == len(text) * n

# Hypothesis strategies generate more inputs for
# the function created by the LLM, helping to
# better explore the function's input space
```

WHAT IT DOES

Hypothesis samples many **diverse inputs** from each strategy and runs the property on all of them, shrinking any failure to a minimal case.

WHY PHASE 2 NEEDS IT

Using Hypothesis automatically surfaces the **edge cases** a single hand-written test misses — empty strings, huge counts, unicode.



Together: Hypothesis drives, Beartype guards

Valid inputs in, contract enforced — and program crashes confirm a definite false negative!



```
from hypothesis import given, strategies as st
from beartype import beartype

@beartype
def repeat(text: str, n: int) -> str:
    return text * n

@given(text=st.text(), n=st.integers(min_value=0))
def test_repeat(text, n):
    repeat(text, n)    # 100s of type-conformant calls
```

CRASH → PHASE 2a · DEFINITE FALSE NEGATIVE

Inputs were type-conformant
(@beartype confirmed) yet the code
crashed → proven type bug. **A**
checker that said "ok" is WRONG.

SUCCESS → PHASE 2b · LIKELY FALSE POSITIVE

Every valid input ran without error →
success witness recorded. **A**
checker that flagged an "error" is
likely WRONG when tests cover.



Phase 2 of Pytifex

Property-based testing with Hypothesis — Detects false negatives and may also find false positives

Type Annotations `def fn(x: int, y: str) -> bool`

HYPOTHESIS

@BEARTYPE

Generate Type-Conformant Inputs

Hypothesis builds inputs from annotations · @beartype enforces conformance

Execute with Generated Inputs

hundreds of calls across type-conformant inputs

CRASH

PHASE 2a

False Negative

Inputs were type-conformant · crash = proven type bug. **Checker said "ok" = WRONG**

SUCCESS

PHASE 2b

Likely False Positive

Execution completed without error · success recorded. **Checker said "error" = wrong?**



A disagreement Pytifex automatically found

Assign a `str` to a `Literal` — three checkers object, one waves it through, which is not good!



```
# simplified code example
UserStatus = Literal['active', 'inactive']

dynamic_status: str = "active"

# assign a plain str into a Literal slot
test: UserStatus = dynamic_status
```

```
$ mypy
error: Incompatible types in assignment
      (str → Literal['active', 'inactive'])
```

mypy

error

CORRECT

pyrefly

error

CORRECT

zuban

error

CORRECT

ty

pass

WRONG

Three checkers reject the unsound assignment; **ty** accepts it — a real **false negative** that Pytifex surfaced automatically!

Is a new release of a type checker always better?

The same file, the same tool — an older pyrefly version caught a real bug that a newer version no longer flags

 example.py

```
class ReportConfig(TypedDict):  
    report_name: str  
    metadata: MetadataDict  
    version: str  
  
def generate_report(**kwargs:  
    Unpack[ReportConfig])  
    if 'extra_config' in kwargs:  
        print(kwargs['extra_config'])  
        # 'extra_config' is not a key
```

Accessing **'extra_config'** — a key `ReportConfig` never declares, which is a type error that a checker should detect

pyrefly 0.44.2 ✓ correct

1 error — bad-typed-dict-key
Flags line 7: 'extra_config' is not a valid key.

pyrefly 0.64.0 ✗ regressed

No errors — silently accepts it
The bug the older version caught now slips through.

Cross-checked — who else catches it?

mypy 2.0.0 1 Error · zuban 0.7.2 1 Error

Illustrates how Pytifax automatically supports DIFFERENTIAL TESTING ACROSS TYPE CHECKER VERSIONS



What to take home about the Pytifex tool?

Is it possible to automatically generate Python programs that cause type checker disagreement? Yes!

Pytifex can automatically generate novel Python programs that can catch type checker disagreements, finding real problems in production type checkers



“You're right that Pyrefly does not implement this currently. [...] We realized that neither mypy nor Pyright check for correct typevar use in classes, which we found surprising. [...] We should error at the definition site, not at the use, as both mypy and pyright do.”

Reference: <https://github.com/facebook/pyrefly/issues/1343>

Advertisement for an upcoming PyOhio 2026 talk!

Testing the Parts You Don't See

Pseudo-Tested Native Code in Python Libraries



"look inside"

Please attend this presentation by **Issei Hasegawa** in
Ballroom D on Sunday, July 26 at 2:45 PM. See you there!



Does it work? Pytifex finds real disagreements!

Across **5 runs**, Pytifex turned **79 generated programs** into **50 checker disagreements**

79

programs generated

50

disagreements found

63%

disagreement rate

4

type checkers compared

How often each checker rejected the code that caused a disagreement (of the 50 disagreements)

