

Differential Inline Testing: Framework, Test Generation, and Application

Chaitanya Shahane, Derek Hansen, Wenxi Wang
The University of Virginia
{ckp6ac, hkk6pa, wenxiw}@virginia.edu

Pengyu Nie
University of Waterloo
pynie@uwaterloo.ca

Abstract—Inline testing was proposed to check the correctness of hard-to-reach code statements, enabling developers to write fine-grained regression tests co-located with the code under test. However, the current inline testing framework requires explicit assertions as test oracles, which can be *brittle*, *buggy*, or *flaky* if not carefully curated—the well-known “oracle” problem. This problem is especially pronounced in domains such as deep learning, where alternative oracles such as differential testing are frequently used.

We propose differential inline testing, which extends the inline testing framework with an implicit, oracle-free mechanism based on differential testing. Instead of specifying an explicit expected output, a differential inline test declares one or more configuration variables (e.g., the hardware backend) and an output variable; the framework then executes the target statement under each configuration and checks for divergences in execution outputs. The resulting tests are persistent regression tests that can be committed to the codebase, and require less maintenance than explicit assertions as the code evolves. We also design and implement DIFFTESTGEN, an LLM-based framework for automatically generating differential inline tests. We evaluate the effectiveness of differential inline testing and DIFFTESTGEN on the codebases of two popular deep learning libraries: PyTorch and TensorFlow. Using DIFFTESTGEN, we generate 9,611 differential inline tests with a validity rate of 92.9%, discovering 175 cross-device divergences. While not every divergence is a bug, we inspected them and reported 6 representative cases to PyTorch developers, 4 of which have been confirmed as previously unknown bugs. The generated differential inline tests also serve as low-maintenance regression tests: no false positives were observed in the evaluated version transitions, compared with 10 false positives from equivalent explicit-oracle tests.

Index Terms—Inline Testing, Differential Testing, Large Language Models, Deep Learning

I. INTRODUCTION

Inline testing [17], [18] was recently proposed as a new testing granularity that enables developers to write fine-grained regression tests for individual program statements. An inline test is co-located with its target statement in the source code, making it easy to write, maintain, and comprehend. As illustrated in Fig. 1a, an inline test directly follows the target statement, and checks the target statement’s output against the expected value given certain test inputs. Prior work has demonstrated that inline tests can detect single-statement bugs [12]–[14] that are missed by unit tests [16], [17].

However, the current inline testing framework relies on explicit assertions (e.g., `check_eq`) as test oracles. Such assertions can be *flaky* [19] when the target statement is non-

deterministic (e.g., random number sampling), or *brittle* [11] when the output value contains components that frequently change across code versions (e.g., numerical precision, internal representations). Automatically extracted assertions during the execution of code or unit tests [16] also suffer from these issues. The difficulty of specifying a correct and robust assertion, well known as the *test oracle problem* [1], remains a barrier to wider adoption of inline testing, especially in domains such as deep learning where outputs vary across hardware backends and library versions [22], [28].

Differential testing [10], [20] is a well-established *oracle strategy* that sidesteps the need for explicit expected outputs: it feeds the same test inputs to multiple configurations of a program (e.g., different hardware backends) and asserts that their outputs should be consistent. Differential testing has been effective at finding deep semantic bugs in compilers [4], [15], [30], [32] and deep learning libraries [23]. However, existing differential testing approaches operate at the program or function level, and cannot directly target individual statements buried inside complicated program logic. Reaching such statements with conventional unit tests or helper functions requires either extracting them into standalone functions solely for testing, or adding print-style instrumentation [17]; both duplicate or restructure production code for the sake of testing, hurting code quality and maintainability. Inline testing instead co-locates the test with the statement under test, providing statement-level targeting without such restructuring.

In this work, we propose *differential inline tests*, which extend the inline testing framework with differential testing as an implicit, oracle-free mechanism. In a differential inline test, developers provide the test inputs and a *configuration variable* that defines the configuration to compare (e.g., the backend device). Developers do *not* specify the expected output. Instead, the framework automatically executes the target statement under each configuration and checks whether the outputs are consistent; any divergence indicates a potential differential bug. By inheriting the co-location and regression properties of inline tests, differential inline tests serve as *persistent regression tests that require little maintenance*: they stay in the codebase, run in CI, and automatically re-verify cross-configuration consistency as the code evolves—without requiring developers to update oracles. We implement differential inline tests as an extension to `pytest-inline` [18].

The power of differential inline tests is amplified when

combined with automated test generation. Since no explicit expected output is required, any generated test input can be used to compose a differential inline test. We design and implement DIFFITESTGEN, an LLM-based framework for generating bug-revealing differential inline tests for given target statements. DIFFITESTGEN consists of two stages: (1) generation, which generates bug-revealing test inputs with code context and API documentation; (2) refinement, which iteratively repairs invalid tests based on execution feedback.

To demonstrate the effectiveness of differential inline testing and DIFFITESTGEN, we evaluate on the codebases of two popular deep learning libraries, PyTorch and TensorFlow, with CPU vs. GPU as the configuration axis. Deep learning libraries are ideal subjects for differential inline testing: their codebases contain many complex tensor operations that should produce consistent results across hardware backends (e.g., CPU vs. GPU), yet are deeply nested in internal modules where traditional differential testing cannot easily reach. We apply DIFFITESTGEN to generate differential inline tests for tensor assignment statements, and evaluate both their bug-finding capability and their value as regression tests across library versions. DIFFITESTGEN generates 8,757 passing differential inline tests from 19,073 target statements, discovering 175 cross-device divergences. The generated differential inline tests also serve as low-maintenance regression tests: no false positives were observed across the two evaluated minor-version transitions, while explicit-oracle tests yield 10 false positives. Since a divergence may or may not indicate a bug, we inspected all of them, grouped them by API, and reported 6 representative GitHub issues to PyTorch developers; so far, 4 have been confirmed as previously unknown bugs and 2 as intended behavior.

The main contributions of this work include:

- **Idea.** We propose differential inline tests, introducing differential testing to inline tests to provide an oracle-free, regression-oriented mechanism for statement-level testing.
- **Testing Framework.** We design and implement differential inline tests as a natural extension to pytest-inline, with configuration variables and library-specific test fixtures.
- **Generation Framework.** We design and implement DIFFITESTGEN, an LLM-based framework for automatically generating differential inline tests.
- **Evaluation.** We evaluate differential inline tests and DIFFITESTGEN on the codebases of PyTorch and TensorFlow, generating 8,757 passing differential inline tests that discover 175 cross-device divergences and serve as low-maintenance regression tests in the evaluated minor-version transitions.

Our code and data are open-sourced at <https://doi.org/10.5281/zenodo.19337365>

```
1 import torch
2 from inline import itest
3 def roll(a:TensorLikeType, shifts:DimsType,
4         dims:DimsType=(0,)) -> TensorLikeType:
5     ...
6     if len_shifts != 1 or len_dims != 1:
7         ...
8         first_dim_rolled = torch.roll(a, (shifts[0],), dims[0])
9         itest().given(a,torch.tensor([[1.,2.],[3.,4.]])
10            .given(shifts,torch.tensor([1]))
11            .given(dims,torch.tensor(0))
12            .check_eq(first_dim_rolled,
13            torch.tensor([[4.,1.],[2.,3.]])
14            return roll(first_dim_rolled, tail_shifts, tail_dims)
```

(a) The target statement (line 7) performs a complex tensor manipulation and is deeply nested in the code (thus hard to be unit-tested). The inline test (line 9) checks the correctness of the target statement by comparing the output with an expected value. Note that the expected value ([[4., 1.], [2., 3.]]) , extracted by executing code on PyTorch 2.9, recorded a buggy behavior on PyTorch 2.9 and fails as a false alarm on PyTorch 2.10.

```
1 def test_differential_torch_roll():
2     def _temp(a, shifts, dims):
3         return torch.roll(a, (shifts[0],), dims[0])
4     a = torch.tensor([[1., 2.], [3., 4.]])
5     shifts = torch.tensor([1])
6     dims = torch.tensor(0)
7     out_cpu = _temp(a, shifts, dims)
8     out_cuda =
9         _temp(a.to("cuda"), shifts.to("cuda"), dims.to("cuda"))
10    assert torch.allclose(out_cpu, out_cuda.to("cpu"))
```

(b) An example differential test for the target statement (using the pytest framework), which requires duplicating the target statement in a temporary function.

```
1 itest().fixture("torch")
2 ...
3 first_dim_rolled = torch.roll(a, (shifts[0],), dims[0])
4 itest().diff_given("device", ["cpu", "cuda"])
5     .given(a,torch.tensor([[1.,2.],[3.,4.]])
6     .given(shifts,torch.tensor([1]))
7     .given(dims,torch.tensor(0))
8     .diff_test(first_dim_rolled)
```

(c) A differential inline test for the target statement. Unlike the inline test in Fig. 1a, the differential inline test checks an equivalence property that should hold regardless of PyTorch version.

Fig. 1: Motivating example comparing inline test, differential test, and differential inline test. The target statement is from PyTorch’s `torch.roll` function¹.

II. BACKGROUND AND MOTIVATION

A. Inline Tests with Explicit Test Oracles

Suppose we are given a function such as the one in Fig. 1a, and want to ensure the correctness of a critical target statement through testing, such as line 7. The desire for testing individual statements is supported by the inline testing framework [17], [18]: complicated code statements frequently introduce single-statement bugs [12]–[14], but testing individual statements in prior testing frameworks would require extracting the target statement into a separate function, introducing additional complexity and maintenance overhead.

An inline test (Fig. 1a, line 9) is a statement inserted into the program to check the correctness of its preceding statement, i.e., the target statement (Fig. 1a, line 7).

1) *Syntax*: Inline tests are written in a *domain-specific language* embedded in the host programming language, where its syntactic components should follow the grammar defined in Fig. 4a.² We introduce the syntactic components using the example in Fig. 1a:

- *declare clause*: a constructor call (`itest()`) that signals the start of an inline test.
- *assign clause*: specifying the values of test input variables (e.g., `given(a, torch.tensor([[1., 2.], [3., 4.])))`).
- *assert clause*: specifying the expected value of the test output variable (e.g., `check_eq(first_dim_rolled, torch.tensor([[4., 1.], [2., 3.])))`).

2) *Execution Workflow*: Note that the inline tests are *not* executed during regular program execution: the `itest()` constructor call and follow-up clauses are intentionally implemented as no-op methods, thus do not incur runtime overhead when they should not. The inline tests are only executed during the testing phase, i.e., when executing the `pytest` with the `pytest-inline` plugin enabled.

Fig. 2 illustrates the three-phase execution workflow of inline tests in `pytest-inline`:

- *Finder*: scans the project’s Python files for `itest()` calls and extracts each inline test with its target statement.
- *Parser*: transforms each inline test from the domain-specific language into an executable test case following the *arrange-act-assert* pattern; namely, the assign clauses are transformed into variable assignments (*arrange*), the target statement is copied verbatim (*act*), and the assert clause is transformed into an assertion (*assert*).
- *Runner*: executes the parsed test and reports pass/fail result.

3) *Challenge: Explicit Test Oracles are Brittle, Buggy, or Flaky*: One limitation of the current inline testing is that an explicit test oracle must be provided in the form of expected output, which introduces several issues. The expected output needs to be carefully curated such that it checks exactly what the code under test is supposed to do. Curating them manually would be tedious, and one common approach is to record the output from executing the code under test and use it as the expected output. For example, in Exli [16], the explicit oracles of inline tests are extracted from the recorded outputs of existing unit tests. However, the recorded output can be a non-trivial object with some of its attributes frequently changing between code versions (e.g., timestamp, random seed, etc.), which introduces *brittle* test oracles that are hard to maintain [11]. Worse, the recorded output may unknowingly encode a bug: in our example, the expected

output for `first_dim_rolled` recorded under PyTorch 2.9.0, `torch.tensor([[4., 1.], [2., 3.]])`, is actually a buggy one³; the bug was fixed in PyTorch 2.10.0, which means the inline test would fail as a false alarm on PyTorch 2.10.0. Another inherent issue of explicit oracles is *flakiness* [19], when the code under test is nondeterministic (e.g., using random numbers).

This facet of the test oracle problem [1] can be addressed via implicit oracles such as differential testing [20].

B. Differential Testing

Differential testing [10], [20] is an implicit oracle strategy that addresses the test oracle problem [1]: instead of requiring an explicit expected output, it asserts that the execution results should be consistent when the same code is run under different configurations. Differential testing is suitable when a code can be executed equivalently under different conditions, for example, executing a tensor operation in deep learning libraries on different backend devices (CPU and GPU), or compiling a program with different compilers or optimization flags [4], [15], [30], [32].

1) *Execution Workflow*: As illustrated in Fig. 3, a differential test executes the same code (test inputs + target statement) under each configuration (e.g., CPU and GPU) and compares the outputs. When the outputs are consistent, the test passes; when they differ, a divergence is reported.

2) *Challenge: Writing Differential Tests as Regression Tests Introduces Code Smells*: Differential testing is traditionally combined with fuzzing [6], [9], [23], [25], [31], [33], where test inputs are generated on-the-fly and discarded; in other words, differential testing is usually used as a one-shot campaign rather than persistent regression tests. However, some differential tests need to be repeatedly conducted on each code version; we observe this practice in real-world test suites such as PyTorch’s `test_randint_randomness_for_large_range`⁵, which compares the behavior of `torch.randint` across CPU and GPU. As a head-to-head comparison, Fig. 1b shows a differential regression test for our motivating example, written following the style of this PyTorch test. To write such a test, developers must introduce a “test harness” that makes the target statement’s output externally observable. This harness either *duplicates* the code under test (code duplication smell) or *refactors* the code to expose intermediate functions solely for testing (a test-induced design change). Either way, it must be updated whenever the target statement changes.

¹https://github.com/pytorch/pytorch/blob/v2.9.0/torch/_refs/_init__.py#L3977

²Between different host programming languages (Python and Java), there are minor differences of naming conventions of inline test’s syntax components (e.g., `itest()` in Python and `new ITest()` in Java); in this work, we stick to the Python naming convention as defined in `pytest-inline` [18].

³See <https://github.com/pytorch/pytorch/issues/167562>: passing a 0-d tensor as the `dims` argument to `torch.roll` silently ignores the dimension

⁴This is a bug we found in our experiments, reported to and confirmed by PyTorch developers. We ask the reviewers not to actively search for this bug in PyTorch’s issue tracker to avoid breaking double-blindness.

⁵https://github.com/pytorch/pytorch/blob/v2.9.0/test/test_cuda.py#L1516-L1539

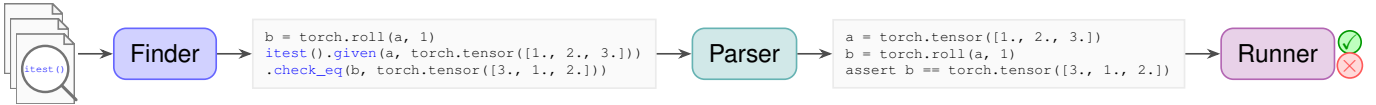


Fig. 2: The workflow of the pytest-inline framework, showing how inline tests are discovered, parsed, and executed.

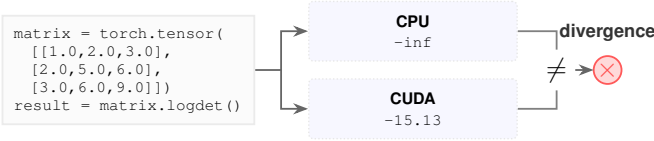


Fig. 3: Example of differential testing: a divergence where `logdet` on a singular matrix returns `-inf` on CPU but a finite value on GPU⁴.

```

1 itest_suite := (tgt_stmt itest+)+
2 itest := declare assign* assert*
3 declare := "itest"((param = val,)*
4 assign := ".given"(input, val)
5 assert := assert_2op | assert_1op | assert_0op
6 assert_2op := ".check_eq" | ".check_neq" | ".check_same" |
7   ".check_not_same" (output, val)
8 assert_1op := ".check_true" | ".check_false" | ".check_none" |
9   ".check_not_none" (output)
10 assert_0op := ".fail"()

```

(a) Grammar for inline tests

```

1 diffitest_suite := [diff_fixture] (tgt_stmt diffitest+)+
2 diff_fixture := "itest().fixture"(library_name)
3 diffitest := declare diff_assign assign* diff_assert
4 diff_assign := ".diff_given"(config_var, val_list)
5 config_var := input_config_var | env_config_var
6 diff_assert := ".diff_test"(output_var)

```

(b) Extended grammar for differential inline tests.

Fig. 4: Grammars

Key Insight. Inline testing and differential testing are naturally complementary. Differential testing provides *implicit* oracles that free inline tests from brittle expected values (§II-A3), while the inline test’s target statement already serves as the test harness, eliminating the code duplication that plagues traditional differential regression tests (§II-B2). We unify them as *differential inline tests*.

III. DIFFERENTIAL INLINE TESTS

We design and implement differential inline testing as an extension to the inline testing framework [18]. Our extension is generic and can be applied to any application scenario of differential testing by providing a library-specific fixture (§III-A3), and our implementation instantiates an important application: testing deep learning libraries’ consistent behavior across backend devices (e.g., CPU and GPU). Fig. 1c shows an example of a differential inline test which we reference throughout this section.

A. Syntax

Differential inline tests extend vanilla inline tests with minimal syntactic additions to support differential testing, while preserving the defining characteristic of inline testing: each test is co-located with and immediately follows its target

statement. Fig. 4b summarizes the extended grammar, which introduces three new constructs on top of the vanilla inline test grammar (Fig. 4a): a fixture clause for configuring library-specific test environments, a diff-assign clause for specifying configuration variables whose values vary across configurations, and a diff-assert clause for designating the output to be compared.

A differential inline test suite (Fig. 4b, line 1) consists of an optional fixture followed by one or more target statements, each accompanied by one or more differential inline tests. Each individual differential inline test (line 3) reuses the declare and assign clauses from vanilla inline tests and adds the diff-assign and diff-assert clauses.

1) *Differential Assignment Clause:* The diff-assign clause (Fig. 4b, line 4) specifies a *configuration variable* whose values vary across executions of the target statement. Each clause binds a single configuration variable to a finite list of values, indicating that the target statement should be executed once per value. A configuration variable (line 5) may be either an *input configuration variable*, such as an algorithm selector that appears as a program-level variable, or an *environment configuration variable*, such as the execution device or library version, drawn from a predefined vocabulary recognized by the framework. For instance, in Fig. 1c, `.diff_given("device", ["cpu", "gpu"])` declares “device” as an environment configuration variable with two values.

The grammar requires each value list to contain at least two elements, ensuring that every differential inline test exercises more than one execution configuration. When multiple diff-assign clauses are present, the configuration space is defined as the Cartesian product of all value lists, allowing a single differential inline test to express multi-dimensional differential comparisons compactly.

2) *Differential Assertion Clause:* Instead of specifying an explicit expected output, a differential inline test concludes with a diff-assert clause (Fig. 4b, line 6). This clause designates the output variable whose values are compared across configurations. For example, `.diff_test(first_dim_rolled)` in Fig. 1c designates the output of the target statement for cross-device comparison. The test passes if the output exhibits consistent behavior across all configurations under the equivalence criterion, and reports a divergence otherwise.

3) *Fixture:* The fixture clause (Fig. 4b, line 2) configures the testing environment at the suite level using library-specific logic. A fixture is a small Python class, written once per library and reused by all differential inline tests in that suite. It subclasses the `DiffITestFixture` class defined in our framework, and implements a fixed interface, namely the methods `setup`, `teardown`, `clone`, `to_device`,

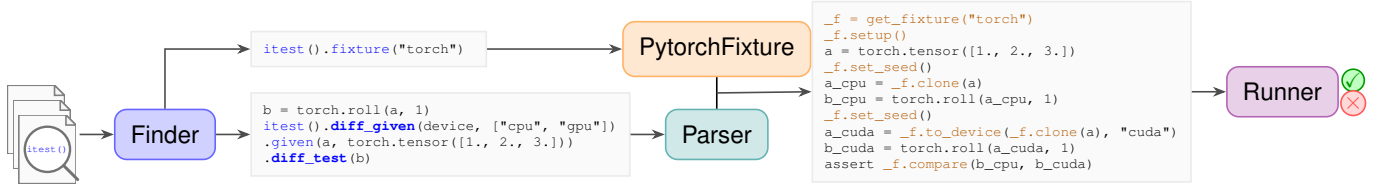


Fig. 5: The workflow of differential inline tests, with the fixture modifying the parser’s behavior.

set_seed, and compare; methods left unimplemented (e.g., teardown) default to no-ops inherited from the base class. Fig. 6 shows the complete PyTorch fixture, which is the only manual artifact required to support a new library. Expressed as `itest().fixture(library_name)`, the clause applies to all subsequent differential inline tests in the suite when present. A fixture encapsulates this logic through three responsibilities:

- *Test setup and teardown.* The fixture specifies per-configuration setup actions (e.g., moving data to a target device, resetting random seeds to control stochasticity) and teardown actions (e.g., restoring state to cleanly isolate subsequent configurations).
- *Customized parser.* The fixture alters inline testing framework’s parser, i.e., how the inline test and target statement are parsed into an executable test case: (1) duplicating the assignments and target statement for each configuration; (2) applying the configuration variable’s value in each configuration, either via simple variable assignment (if input configuration variable) or code structure manipulation (if environment configuration variable). Crucially, the parser remains library-agnostic: instead of inlining any library-specific operation, it emits *calls* to the fixture’s interface methods (e.g., `_f.clone(...)`, `_f.to_device(...)`, `_f.compare(...)` in Fig. 5), and each library’s fixture supplies the concrete implementation. This is why, for example, `to_device` appears in the generated test as the interface call `_f.to_device(...)` rather than a hardcoded `.to(device)`.
- *Test output equivalence criterion.* The fixture specifies how outputs from different configurations are compared. The default is exact equality (`==`), but domain-specific fixtures can override this with tolerance-aware comparison (e.g., `allclose` for floating-point tensors).

B. Execution Workflow

Fig. 5 illustrates the execution workflow, which extends the vanilla inline test workflow (Finder, Parser, Runner; Fig. 2) with a fixture-modified parser. The *Finder* additionally extracts the fixture clause if present, and the framework loads the corresponding fixture (e.g., `TorchDiffITestFixture` for `itest().fixture("torch")`). The *Parser* transforms the differential inline test into an executable differential test supported by the fixture: it sets up the fixture (`_f`), duplicates the assignments (arrange) and target statement (act) per configuration, and calls `_f.compare()` to check output consistency (assert). Notably, the transformation remains library-agnostic, and all library-specific operations are encapsulated in the fixture’s methods (e.g., `_f.clone()` for input cloning, `_f.to_device()` for device placement). The *Runner* executes

```

1 class TorchDiffITestFixture(DeepLearningLibFixture):
2     def setup(self):
3         os.environ["CUBLAS_WORKSPACE_CONFIG"] = ":4096:8"
4     def clone(self, v): return v.clone()
5     def to_device(self, v, d): return v.to(d)
6     def set_seed(self, seed=42):
7         random.seed(seed); torch.manual_seed(seed)
8         np.random.seed(seed)
9     def compare(self, a, b):
10        a, b = a.detach().cpu(), b.detach().cpu()
11        return torch.allclose(a, b, rtol=1e-4, atol=1e-4, equal_nan=True)

```

Fig. 6: Differential inline test fixture for PyTorch.

the test, reporting a divergence when outputs are inconsistent across configurations.

Some of the fixture’s methods will also be invoked during the execution of the differential test. Fig. 6 shows the PyTorch fixture used for our motivating example; this fixture is bundled in our differential inline testing framework. The fixture’s setup method configures the CUBLAS environment variable to make some stochastic tensor operations deterministic; the clone and to_device methods delegate to the corresponding methods in PyTorch. The compare method compares the outputs using `torch.allclose` with tolerances of 10^{-4} (allowing for relative and absolute errors each up to 10^{-4}) and `equal_nan = True` (considering nan values to be equal); these parameters are chosen to minimize the false positive divergence due to minor floating-point differences.

IV. DIFFITESTGEN

To support the application of differential inline testing, we design and implement DIFFITESTGEN, an automated framework for generating bug-revealing differential inline tests. Fig. 7 presents the overall workflow. Given a target statement, the Context Retriever (§IV-A) assembles relevant context (API documentation, surrounding code, etc.) that guides test generation. The Generation Agent (§IV-B) is instructed to generate differential inline tests that are likely to trigger divergences while being suitable for inclusion as regression tests. The generated tests are executed by the differential inline testing framework (§III-B), whose outcomes are classified as *Passed* (consistent outputs across configurations), *Divergent* (inconsistent outputs across configurations), or *Crashed* (runtime error or timeout). Passed tests are collected into a regression test suite, while divergent tests should be investigated for potential bugs. Crashed tests, along with their error logs, are sent to the Refinement Agent (§IV-C) for iterative repair; corrected tests are re-executed and re-classified, forming a feedback loop that steadily improves test quality.

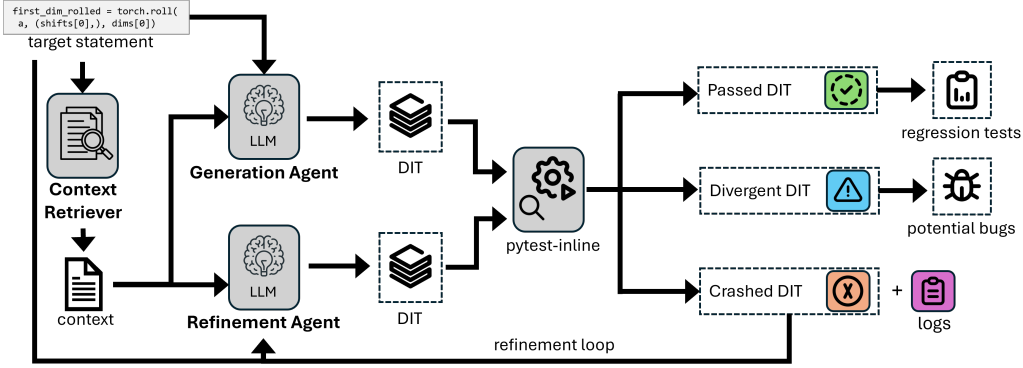


Fig. 7: The overall workflow of DIFFITESTGEN

DIFFITESTGEN is designed to be configurable with respect to the underlying foundation LLM and the number of refinement iterations. In this work, we use OpenAI’s GPT-5.4 and run refinement up to 3 iterations.

A. Context Retriever

The Context Retriever assembles the information that enables downstream agents to generate high-quality differential inline tests. For each target statement, it retrieves:

- *Surrounding code context*: the enclosing function and class, import statements, and preceding statements.
- *Type annotations*: input and output types inferred by a type checker (pyrefly [24]), enabling the agents to construct test inputs of the correct types.
- *API documentation*: the documentation of the API called by the target statement, guiding the generation of meaningful and bug-revealing test inputs.
- *Stochastic constraints*: some APIs are inherently stochastic but can be made deterministic under some constraints (e.g., `torch.nn.RNN` can be made deterministic by setting `dropout = 0`); these constraints, which should be provided by the user, are useful to reduce false positive divergences.

B. Generation Agent

The Generation Agent takes the target statement and enriched context and generates up to N differential inline tests per target statement ($N=10$ by default). To balance creativity with reliability, we use a temperature of 0.8 with nucleus sampling ($\text{top-p}=0.95$); if the first attempt yields no test, a second attempt is made at a lower temperature of 0.3. The prompt of the Generation Agent is organized into three parts:

- *Common instructions*: introduces the grammar of differential inline test, the input format (target statement and context), the output format (sample differential inline tests), and guidelines towards generating valid differential inline tests.
- *Domain-specific instructions*: this part should be customized to the target differential testing scenario, providing guidance on what tests to generate and avoid; for example, for differential testing of deep learning libraries on different devices, test inputs should generally be small tensors that can contain corner-case values, but stochastic operations and single-device operations should be avoided.

- *Library-specific instructions*: this part should be customized to the specific library/codebase containing the target statement, providing concrete examples of differential inline tests and common test inputs (such as tensors in PyTorch), following the idea of in-context few-shot learning.

The agent is explicitly instructed to return an empty response, effectively skipping generation, when the target statement is unsuitable for differential testing. Specifically, it skips statements that (1) require a special runtime context that is unavailable (e.g., distributed process groups), (2) invoke stochastic operations and cannot be made deterministic, (3) produce outputs that cannot be meaningfully compared (e.g., non-tensor outputs are not supported by the fixture for PyTorch and other deep learning libraries). This prompt-level filtering reduces false positives without requiring manual filtering of target statements.

C. Refinement Agent

Whenever a generated differential inline test crashes during execution, the Refinement Agent iteratively repairs it. The agent receives the crashed test code together with its error log (error type and message), and produces a corrected version that preserves the original test intent while resolving the identified issue. To prioritize correctness over diversity, it uses a lower temperature of 0.3 (compared to 0.8 for the Generation Agent). The prompt of the Refinement Agent is organized into multiple parts: common instructions, domain-specific instructions, library-specific instructions, and *error-specific instructions*. The first three parts are similar to the Generation Agent. The last part is triggered when the error type matches a known failure pattern, in which case the error-specific instructions provide a targeted fix (e.g., for out-of-memory errors, the instructions may suggest using smaller tensors or reducing the batch size).

V. APPLICATION: DIFFERENTIAL INLINE TESTING FOR DEEP LEARNING LIBRARIES

To evaluate the effectiveness of DIFFITESTGEN and differential inline tests, we apply them to the codebases of two popular deep learning libraries: PyTorch and TensorFlow. Our evaluation focuses on the regression testing setup: DIFFITESTGEN generates differential inline tests for target statements found in the libraries’ source code, and we measure their

ability to reveal cross-device divergences and detect cross-version regressions. We structure our evaluation around the following research questions:

RQ1. How effective is DIFFITESTGEN at generating valid differential inline tests, and do they reveal cross-device divergences in the codebases of PyTorch and TensorFlow?

RQ2. Can differential inline tests serve as regression tests that detect behavioral changes across library versions, and how do they compare to explicit-oracle inline tests for this purpose?

RQ3. What are the common error patterns in the crashed tests generated by DIFFITESTGEN?

A. Subjects

We evaluate on two widely used deep learning libraries: PyTorch (v2.9.0 for RQ1; v2.9.0 and v2.10.0 for RQ2) and TensorFlow (v2.19 for RQ1; v2.19 and v2.20 for RQ2).

To identify target statements suitable for cross-device differential inline testing, we parse every Python file, extract assignment statements, and apply the following filters: (1) statements with no inputs or no tensor-typed inputs are removed; (2) statements referencing `self` or producing structured outputs (tuple unpacking) are discarded; (3) trivial assignments (bare names or constants) are discarded; (4) target statements using always-stochastic or device-irrelevant APIs are excluded. Finally, we deduplicate target statements across the codebase.

a) Stochasticity and Device-Irrelevance API Identification: To identify APIs that may be stochastic or device-irrelevant, we crawl the PyTorch and TensorFlow API documentation and search it for indicative keywords (`{random, sample, stochastic, dropout, uninitialized}` for stochastic, and `{mps, xpu, vulkan, mtia, privateuse, hpu, ipu, tpu}` for device-irrelevant), yielding 387 candidate stochastic and 132 candidate device-irrelevant APIs that the authors collectively reviewed and labeled. We classify each stochastic API as *always-stochastic* (excluded from our study; e.g., `torch.randn`), *conditioned* (kept, with a manually annotated determinism constraint passed to DIFFITESTGEN; e.g., `torch.nn.RNN`), *mitigated* (made deterministic by the fixture’s setup; e.g., `torch.empty`), or *deterministic* (an accidental keyword match); the confirmed device-irrelevant APIs are excluded from our study.

b) Type Inference: We use pyrefly [24] to infer tensor-typed variables for PyTorch; since no type-inference tool worked on TensorFlow, we fall back to tagging variables from known tensor-producing modules.

c) Dataset Statistics: Table I summarizes the filtering results. From 3,900 Python files in PyTorch, the pipeline retained 18,121 target statements (5.8%) across 1,051 source files. From 3,137 files in TensorFlow, it retained 952 statements (0.6%) across 187 files.

B. Experimental Environment

We conduct our evaluation on a machine with an NVIDIA RTX A6000 GPU (48 GB VRAM) and CUDA 13.0. DIFFITESTGEN adopts GPT-5.4 as the foundation LLM, specifically used via the OpenAI API in flex mode. The entire

TABLE I: Dataset and filtering statistics.

Filter Stage	PyTorch	TensorFlow
Total Python files	3,900	3,137
Total assignment stmts	313,091	171,960
No inputs (non-assignment)	115,968	64,912
No tensor inputs (non-tensor type)	102,564	75,133
Self-references (<code>self.*</code>)	23,403	19,319
Structured output (tuple unpacking)	6,445	3,494
Trivial RHS (bare name / constant)	38,309	7,767
Always-stochastic / non-device APIs	398	0
Cross-file duplicates	7,867	375
Final target statements	18,121	952

experiments took about 5 GPU-days to complete. DIFFITESTGEN used 49,963,338 input and 3,784,214 output tokens for generation, and 4,494,784 input and 1,011,951 output tokens for refinement, costing \$104.04 in total (\$0.0055 per target statement).

C. RQ1: DIFFITESTGEN Performance

We evaluate DIFFITESTGEN with two metrics. The *validity rate*, $(\#Pass + \#Diverge) / \#Tests\ Gen.$, is the fraction of generated tests that execute without crashing; it is not bug-finding precision, since a divergence may be intended behavior. *Statement coverage*, $\#Tests\ Gen. / \#Target\ Stmts$, is the fraction of target statements for which the agent does not skip. Each counted test is one target statement and may hold up to $N=10$ inline tests (§IV-B). For refinement, $\#Target\ Stmts$ is the previous stage’s crashed tests, and coverage is how many of those crashes were regenerated.

Table II presents the results of DIFFITESTGEN’s test generation and refinement process on both codebases.

PyTorch. DIFFITESTGEN generated 8,420 passing differential inline tests from 18,121 target statements, achieving a final validity rate of 93.4%. The generation stage produced 7,800 passing tests (validity rate 86.4%, statement coverage 50.8%) and discovered 151 divergences. Through three refinement iterations, the framework recovered 638 additional passing tests, improving the validity rate from 86.4% to 93.4%, and discovered 169 divergences in total across 76 distinct APIs. The refinement showed diminishing returns, suggesting that remaining failures represent complex semantic issues not easily addressed through error-message-guided repair.

TensorFlow. DIFFITESTGEN generated 337 passing differential inline tests from 952 target statements (validity rate 83.3%, statement coverage 43.3%). Refinement recovered 227 additional passing tests, totaling 6 divergences across 3 APIs. Refinement follows a similar diminishing-returns pattern.

Summary. Across both libraries, DIFFITESTGEN generated 8,757 passing differential inline tests (validity rate 92.9%, statement coverage 50.4%) and discovered 175 cross-device divergences; our design intentionally favors validity over coverage to minimize false positives requiring manual inspection.

TABLE II: Test generation and execution results across DIFFITESTGEN’s stages.

	Stage	#Target Stmts	#Skip	#Tests Gen.	#Pass	#Diverge	#Crash	Valid. Rate	Stmt. Cov.
PyTorch	Generation	18,121	8,922	9,199	7,800	151	1,248	86.4%	50.8%
	Refinement-1	1,248	228	1,020	503	17	500	51.0%	81.7%
	Refinement-2	500	53	447	87	1	359	19.7%	89.4%
	Refinement-3	359	22	337	30	0	307	8.9%	93.9%
	DIFFITESTGEN (final)	18,121	8,922	9,199	8,420	169	610	93.4%	50.8%
TensorFlow	Generation	952	540	412	115	1	296	28.2%	43.3%
	Refinement-1	296	34	262	192	5	65	75.2%	88.5%
	Refinement-2	65	7	58	27	0	31	46.6%	89.2%
	Refinement-3	31	2	29	3	0	26	10.3%	93.5%
	DIFFITESTGEN (final)	952	540	412	337	6	69	83.3%	43.3%
Total		19,073	9,462	9,611	8,757	175	679	92.9%	50.4%

D. RQ2: Cross-Version Regression Detection

We evaluate whether differential inline tests can serve as low-maintenance regression tests across library versions. Each test still checks CPU vs. GPU (`diff_given("device", ["cpu", "gpu"])`); the library version is the external axis on which we re-execute the same files (PyTorch 2.9.0 $\rightarrow$ 2.10.0; TensorFlow 2.19 $\rightarrow$ 2.20) and classify pass-to-pass, pass-to-diverge, pass-to-crash, and diverge-to-diverge outcomes. For each differential inline test that passes on $v(N)$, we also build a hypothetical explicit-oracle inline test from the recorded output and re-run it on $v(N+1)$. A false positive is such a test that fails on $v(N+1)$ while the differential inline test still passes; the comparison covers only tests that pass on $v(N)$.

Results. For PyTorch, of the 9,065 tests with unchanged target statements (134 excluded), 8,299 passed on v2.9.0: 7,688 still pass, 1 newly diverges, and 610 crash due to API changes; 149 tests that already diverged still do. For TensorFlow, of the 337 tests that passed on v2.19, 333 still pass and 4 newly diverge; 6 diverge on both versions and 69 crash on both. 8 PyTorch and 2 TensorFlow explicit-oracle tests failed on the new version while the corresponding differential inline tests still passed; all are false positives from brittle or buggy oracles. No false positives were observed for differential inline tests in these two minor-version transitions, versus 10 for explicit-oracle tests. The newly divergent tests (1 PyTorch, 4 TensorFlow) are actual cross-device output differences on the new version and remain to be investigated; we count them neither as false positives nor as bugs. Major-version API changes, as in the pass-to-crash outcomes, can still require maintenance, so differential inline tests reduce it but do not eliminate it.

E. RQ3: Error Analysis

Fig. 8 shows the distribution and progression of error categories across DIFFITESTGEN’s generation and refinement stages for both libraries. The error analysis reveals two key findings. First, *context limitation* is the primary bottleneck: across both libraries the dominant crashes are `NameErrors`, where the LLM cannot resolve identifiers defined in distant scopes or deeply nested internal modules that are not present in the provided context (most acute for TensorFlow). This

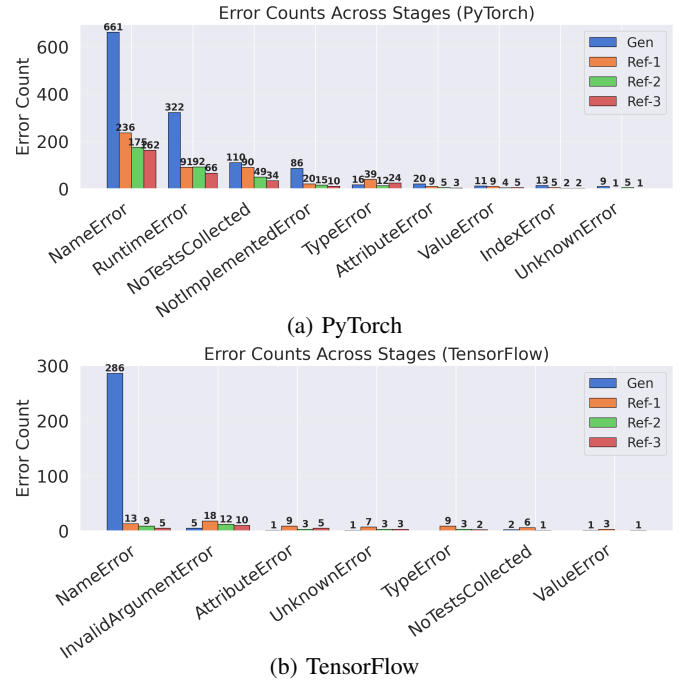


Fig. 8: Error pattern progression across iterations.

suggests that richer context retrieval (e.g., project-wide symbol resolution) could substantially improve test generation. Second, refinement helps most when the error message is informative (e.g., `RuntimeError`) and less when the failure is missing context (`NameError`).

VI. DISCUSSIONS

A. Real-World Impact

We inspected all 175 discovered divergences and grouped them by the API they exercise (79 APIs), so related cases are reviewed together and the developer’s review effort is minimized. Expected differences arise from two sources. First, stochastic operations are inherently nondeterministic across devices: as PyTorch developers note in a GitHub issue⁶, each device uses a different random-number-generator implementation, so identical seeds yield different sequences (e.g., `torch.multinomial`, `torch.nn.Dropout`); we therefore filter

⁶<https://github.com/pytorch/pytorch/issues/2976>

out known stochastic operations before generation. Second, some statements legitimately differ by design (e.g., eigendecomposition may return eigenpairs in different orders); we prompt the LLM to skip such APIs. Among the remaining groups we prioritized qualitatively different outputs (e.g., $-\text{inf}$ vs. a finite value) over small numerical noise, and filed 6 representative PyTorch issues: 4 were confirmed as previously unknown bugs and 2 as intended design.

Adopting differential inline tests in a codebase requires adding the inline scaffolding next to target statements plus one library-specific fixture (Fig. 6). We argue this cost is modest and controllable. Deploying on a new library is a one-time cost: that fixture, a review of candidate stochastic and device-irrelevant APIs (387 and 132 in our study), library-specific prompts, and divergence triage. It is paid once per library, not once per test, and can be potentially automated with LLM-assisted API labeling. The inline clauses are no-ops during normal execution so they add no runtime overhead, and only run under PyTest with the plugin enabled [17]. A project need not annotate every statement: differential inline tests can be added incrementally to the most critical or bug-prone statements, and DIFFTESTGEN helps by generating them automatically. While adding differential inline tests could impact readability, it can be mitigated by folding the test lines in IDEs/editors.

Building on these findings, we plan to integrate the differential inline test framework into the official pytest-inline repository (so any Python project can adopt it with minimal setup), and to deploy a curated set of generated differential inline tests into PyTorch and TensorFlow to gather developer feedback on their usefulness in continuous integration. Future work includes extending the approach to JAX and other libraries, and studying longer and major-version histories.

B. Threats to Validity

a) Internal Validity: DIFFTESTGEN relies on LLM agents for both test generation and refinement, so the quality of the generated differential inline tests is inherently bounded by the LLM’s understanding of API semantics, tensor operations, and edge cases. To mitigate prompt sensitivity, we use a structured prompt template with few-shot examples and apply three rounds of error-guided refinement. The underlying LLM is a configurable component, so DIFFTESTGEN is model-independent in its architecture, but we evaluated only GPT-5.4; performance with other models, including open-weight ones, is not empirically established and is left to future work. Our stochastic-API classification (always-stochastic, mitigated, and conditioned) requires a manually curated list; an incomplete or incorrect list could introduce false-positive divergences or miss genuine bugs. We mitigate this by systematically searching API documentation with keywords and having the authors collectively review all candidates.

b) External Validity: Our empirical findings are scoped to PyTorch and TensorFlow with CPU vs. GPU as the configuration axis. Although the differential inline test framework is general-purpose, applying DIFFTESTGEN to other domains

or libraries (e.g., compilers or JAX) would require adapting the target-statement finder, the stochastic-API classification, the library-specific fixture, and the generation/refinement prompts. Other configuration axes and major-version evolution are likewise unevaluated.

c) Construct Validity: We measure the validity rate (the fraction of generated tests that pass or diverge, rather than crash) and statement coverage (the fraction of target statements for which a test is generated) to evaluate test quality, and count divergences to assess bug-finding ability. The validity rate is not bug-finding precision: a divergence may be intended behavior. Divergences are not automatically classified as bugs; we manually inspect all of them and submit representative cases as GitHub issues for confirmation by library developers. The false-positive comparison against hypothetical explicit-oracle inline tests covers only the two evaluated minor-version transitions.

VII. RELATED WORK

A. Inline Testing and Differential Testing

Inline tests [17] target the correctness of individual statements; Exli [16] generates them by recording variable values from existing unit-test executions, but such recorded oracles are prone to the test oracle problem [1]. We instead embed differential testing into inline tests, where the oracle is cross-configuration agreement rather than a hardcoded value.

Differential testing [10], [20] has been especially successful in compiler testing [4], [15], [30], [32] and, in deep learning, in cross-model and cross-backend testing such as DeepXplore [22] and CRADLE [23]. We bring it to a new granularity—inline testing—adding code co-location, statement-level targeting, and persistent regression testing, addressing both the oracle problem and the maintenance of oracles across library versions.

B. LLM-based Test Generation

TitanFuzz [6] and FuzzGPT [7] use LLMs to generate API inputs and unusual programs that trigger edge cases, but they still target crashes in public APIs.

Our work is the first to apply LLMs to differential inline test generation: a two-stage framework (generation and refinement) that uses context retrieval and execution feedback to produce tests for internal statements, where the LLM supplies inputs and the framework supplies the differential oracle.

C. Bug Detection for Deep Learning Libraries

Fuzzers such as FreeFuzz [25], DeepREL [8], DocTer [27], IvySyn [5], and NablaFuzz [29] target crashes in deep learning libraries and discard inputs after a campaign. Differential inline tests instead target internal statements and remain in the codebase as regression tests, so we position them as a complement to fuzzing rather than a replacement.

Metamorphic testing [3] addresses the oracle problem via relations between related executions, and has been applied to deep learning models, operators, and compilers [2], [21], [26], [28], [34]. Extending inline testing to metamorphic relations is future work.

VIII. CONCLUSIONS

We proposed differential inline tests, which extend the inline testing framework with differential testing as an oracle-free, regression-oriented mechanism for statement-level testing: a differential inline test declares configuration variables (e.g., backend device) and an output variable, and the framework executes the target statement under each configuration and checks for divergences without an explicit expected output. We implemented differential inline tests atop pytest-inline with fixtures for PyTorch and TensorFlow, and built DIFFITESTGEN, an LLM-based framework that automatically generates bug-revealing differential inline tests for statements in library codebases. On two deep learning libraries, DIFFITESTGEN generated 8,757 passing differential inline tests from 19,073 target statements, revealing 175 cross-device divergences (4 confirmed as previously unknown PyTorch bugs); as regression tests on the two evaluated minor-version transitions, no false positives were observed, versus 10 for equivalent explicit-oracle tests.

DATA AVAILABILITY

Our code and data are available at <https://doi.org/10.5281/zenodo.19337365>.

ACKNOWLEDGMENT

We thank Yu Liu and the anonymous reviewers for their comments and feedback. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under funding reference RGPIN2024-04909.

REFERENCES

- [1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *TSE*, vol. 41, no. 5, pp. 507–525, 2014.
- [2] J. Chen, C. Jia, Y. Yan, J. Ge, H. Zheng, and Y. Cheng, “A miss is as good as a mile: Metamorphic testing for deep learning operators,” in *FSE*, 2024, pp. 2005–2027.
- [3] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, “Metamorphic testing: A review of challenges and opportunities,” *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–27, 2018.
- [4] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, “Coverage-directed differential testing of JVM implementations,” in *PLDI*, 2016, pp. 85–99.
- [5] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, “IvySyn: Automated vulnerability discovery in deep learning frameworks,” in *USENIX Security*, 2023, pp. 2383–2400.
- [6] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *ISSTA*, 2023, pp. 423–435.
- [7] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, “Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt,” *arXiv preprint arXiv:2304.02014*, 2023.
- [8] Y. Deng, C. Yang, A. Wei, and L. Zhang, “Fuzzing deep-learning libraries via automated relational API inference,” in *FSE*, 2022, pp. 44–56.
- [9] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in *USENIX Workshop on Offensive Technologies*, 2020.
- [10] M. A. Gulzar, Y. Zhu, and X. Han, “Perception and practices of differential testing,” in *ICSE-SEIP*, 2019, pp. 71–80.
- [11] C. Huo and J. Clause, “Improving oracle quality by detecting brittle assertions and unused inputs in tests,” in *FSE*, 2014, pp. 621–631.
- [12] A. V. Kamienski, L. Palechor, C.-P. Bezemer, and A. Hindle, “PySSuBs: Characterizing single-statement bugs in popular open-source Python projects,” in *MSR*, 2021, pp. 520–524.
- [13] R.-M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? the ManySSuBs4J dataset,” in *MSR*, 2020, pp. 573–577.
- [14] J. Latendresse, R. Abdalkareem, D. E. Costa, and E. Shihab, “How effective is continuous integration in indicating single-statement bugs?” in *MSR*, 2021, pp. 500–504.
- [15] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, 2014.
- [16] Y. Liu, P. Nie, A. Guo, M. Gligoric, and O. Legunsen, “Extracting inline tests from unit tests,” in *ISSTA*, 2023, pp. 1458–1470.
- [17] Y. Liu, P. Nie, O. Legunsen, and M. Gligoric, “Inline tests,” in *ASE*, 2022, pp. 57:1–13.
- [18] Y. Liu, Z. Thurston, A. Han, P. Nie, M. Gligoric, and O. Legunsen, “pytest-inline: An inline testing tool for python,” in *ICSE-Demo*, 2023, pp. 161–164.
- [19] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *FSE*, 2014, pp. 643–653.
- [20] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
- [21] Y. Mu, J. Zhai, C. Fang, X. Chen, Z. Cao, P. Yang, K. Zhao, A. Guo, and Z. Chen, “Improving deep learning framework testing with model-level metamorphic testing,” in *ISSTA*, 2025, p. to appear.
- [22] K. Pei, Y. Cao, J. Yang, and S. Jana, “DeepXplore: Automated whitebox testing of deep learning systems,” in *SOSP*, 2017, pp. 1–18.
- [23] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, “CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries,” in *ICSE*, 2019, pp. 1027–1038.
- [24] Pyrefly Team, “Pyrefly: A fast python type checker,” <https://pyrefly.org/>, 2026, accessed: 2026-03-12.
- [25] A. Wei, Y. Deng, C. Yang, and L. Zhang, “Free lunch for testing: Fuzzing deep-learning libraries from open source,” in *ICSE*, 2022, pp. 995–1007.
- [26] D. Xiao, Z. Liu, Y. Yuan, Q. Pang, and S. Wang, “Metamorphic testing of deep learning compilers,” *POMACS*, vol. 6, no. 1, pp. 1–28, 2022.
- [27] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, “DocTer: Documentation-guided fuzzing for testing deep learning API functions,” in *ISSTA*, 2022, pp. 176–188.
- [28] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, “Testing and validating machine learning classifiers by metamorphic testing,” *JSS*, vol. 84, no. 4, pp. 544–558, 2011.
- [29] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, “Fuzzing automatic differentiation in deep-learning libraries,” in *ICSE*, 2023, pp. 1174–1186.
- [30] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” in *PLDI*, 2011, pp. 283–294.
- [31] M. Zalewski, “American fuzzy lop - whitepaper,” 2016. [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt
- [32] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, “Compiler testing using template Java programs,” in *ASE*, 2022, pp. 1–13.
- [33] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. CISA Helmholtz Center for Information Security, 2024. [Online]. Available: <https://www.fuzzingbook.org/>
- [34] Z. Zhang, P. Wang, H. Guo, Z. Wang, Y. Zhou, and Z. Huang, “DeepBackground: Metamorphic testing for deep-learning-driven image recognition systems accompanied by background-relevance,” *IST*, vol. 140, p. 106701, 2021.