Demystifying Deep Learning Compiler Frontend Bugs: An LLM-Aided Empirical Study

Xinyi Yuan
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
yuanxinyi22@otcaix.iscas.ac.cn

Wei Chen*
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
wchen@otcaix.iscas.ac.cn

Jinyi Liu
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
liujinyi24@otcaix.iscas.ac.cn

Pengyu Chen
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
chenpengyu23@otcaix.iscas.ac.cn

Jun Wei
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
wj@otcaix.iscas.ac.cn

Guoquan Wu
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
gqwu@otcaix.iscas.ac.cn

Jiaxin Zhu
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
zhujiaxin@otcaix.iscas.ac.cn

Tao Huang
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
tao@otcaix.iscas.ac.cn

Abstract

Deep learning compilers (DLCs) are designed to translate deep learning programs into optimized, hardware-specific code. Typically, DLC frontends translate programs into graph-based intermediate representations (IRs) to enable optimizations. Defects introduced during this stage (termed *fBug*s) are severe yet understudied, as prior work predominantly focuses on low-level APIs and operators or treats DLCs as monolithic entities.

To bridge this gap, we conduct the first systematic empirical study of *fBug*s in TorchDynamo, the default DLC frontend for PyTorch 2, the most popular DL framework. Leveraging a domain-knowledge-enhanced LLM-aided methodology, we analyze 123 *fBug*s and construct a taxonomy comprising 7 root cause categories and 15 subcategories. Our findings provide actionable insights for DLC development and testing. Furthermore, we leverage the LLM to generate targeted, root cause-aware test cases to detect new bugs. We uncovered 23 previously unknown *fBug*s in recent releases (15 confirmed) across eight (sub)categories, demonstrating the efficacy of our methodology in testing and hardening DLC frontends.

Keywords

Deep Learning Compiler; Frontend Bug; PyTorch; TorchDynamo; LLM; Empirical Study

*Corresponding author.

1 Introduction

The rapid adoption of deep learning (DL) has driven the development of deep learning compilers (DLCs), which bridge the gap between high-level frameworks (e.g., PyTorch [2, 45] and TensorFlow [1]) abstractions and hardware-specific optimizations. DLCs, such as TVM [5], Glow [49], and AKG [72], typically consist of a frontend and a backend. The frontend captures the computational graph by translating high-level DL models into an intermediate representation (IR) and applies graph-level optimizations like operator fusion and constant folding; the backend further lowers the IR into hardware-specific operators, ensuring efficient execution. By decoupling algorithm design from hardware implementation, DLCs enhance both usability and performance, making them critical for the reliability and efficiency of DL frameworks.

However, defects in DLCs can have severe consequences. They may propagate through the compilation pipeline and lead to crashes, incorrect outputs, or performance degradation. While prior studies have tested DLCs and investigated their bugs, most focus on backend components, such as operator implementations or hardware-specific optimizations. The frontend's program-to-graph translation and corresponding errors remain relatively underexplored, despite being equally critical. To bridge this gap, we conduct the first empirical study on the bugs (i.e., ***fBug*s**) of the compiler frontend **TorchDynamo** in the most popular DL framework, PyTorch 2 [2]. We aim to demystify the characteristics of *fBug*s in terms of their distributions, triggers, and root causes.

We propose an LLM-aided empirical study that addresses key challenges in analyzing complex, real-world bug reports. To guide the LLM in extracting critical information from noisy, unstructured bug reports and patches, we design a specialized template to characterize *fBug* features. To mitigate LLM hallucination and improve reliability, we further inject domain knowledge of TorchDynamo as contextual grounding. Building upon the LLM's analysis, we manually review and consolidate the results to distill the root causes of bugs into a fine-grained classification system, which serves as a structured reference for downstream tasks. Leveraging information on bug root causes, an LLM can generate new test cases to detect bugs in the DLC frontend of the recent version.

Following the methodology, our in-depth investigation of 123 *fBug*s identifies 7 root cause categories and 15 subcategories. We obtain several key insights: inadequate support in the DLC frontends for complex Python features, e.g., dynamic typing, metaprogramming, and runtime introspection, is a major source of *fBug*s; *fBug*s sharing the same root cause tend to be triggered by similar Python constructs and runtime conditions; and identified root causes enable the LLM to generate test cases for new *fBug*s. Our LLM-aided method detects 23 *fBug*s in later DLC versions, with 15 confirmed across eight (sub)categories.

The main contributions are summarized as follows:

- We conduct the first systematic empirical study of DLC *fBug*s in the most popular DL framework PyTorch, yielding valuable insights into bug root causes and characteristics.
- We design a novel LLM-aided empirical study methodology, which can (1) analyze *fBug*s accurately and summarize them reasonably based on the instructions enriched with DLC domain knowledge and (2) effectively create test cases, based on root causes, that can trigger previously unseen *fBug*s.
- This work offers valuable takeaways for the development, use, and testing of the DLC frontend. Source code and datasets are publicly available for reproducibility and future research.

The rest of this paper is organized as follows. Section 2 introduces DLCs and PyTorch's compilation pipeline. Section 3 presents our motivation and explains our focus on TorchDynamo. Section 4 outlines our research questions and proposes the LLM-aided methodology. Section 5 details the study results, and Section 6 discusses the role of LLMs, implications, limitations, and threats to validity. Section 7 reviews related work, and Section 8 concludes the paper.

2 Deep Learning Compiler

DLCs bridge high-level programming frameworks and diverse hardware, translating DL models into optimized, hardware-specific code that improves parallelism and memory efficiency. A rich ecosystem of DLCs has emerged, including TVM [5], Glow [49], XLA [57], nvFuser [55], AKG [72], and TorchDynamo with TorchInductor [2].

PyTorch 2 has emerged as a leading DL framework, widely adopted in both academic research and industry for its balance of usability and performance [62]. A key advancement in PyTorch 2 is its staged compilation pipeline, activated via `torch.compile`, which addresses the limitations of prior eager and graph execution modes. The pipeline consists of three core components — *TorchDynamo*, *AOTAutograd*, and *TorchInductor* — each responsible for optimizing execution at a distinct stage. Specifically,

Figure 1: An Issue of PyTorch Related to DLC Frontend

TorchDynamo acts as the DLC frontend. It captures the computational graph by simulating Python bytecode execution and generating an **FX graph** [48] that encodes the forward computation and necessary runtime constraints.

AOTAutograd takes the FX graph as input and applies ahead-of-time automatic differentiation to construct explicit backward computation graphs. Both forward and backward graphs serve as the input to backend compilation.

TorchInductor is the default compiler backend that consumes the forward and backward graphs and lowers them to hardware-specific code. It applies backend-level optimizations such as operator fusion and memory planning, and generates executable kernels for CPUs and GPUs (e.g., via Triton [58] or C++).

3 Motivation and Object

This section analyzes the problem and explains why we target TorchDynamo.

3.1 Problem Analysis

DLCs are crucial to the correctness and performance of DL programs, but some concerns still remain. First, prior studies on DLC bugs often treat DLCs as monoliths, neglecting the distinct roles of the frontend and backend. This obscures stage-specific failure modes and hinders a deeper understanding of bugs. Second, existing work focuses predominantly on backend components, leaving the frontend's program-to-graph translation understudied. However, errors from the frontend are equally critical as they can propagate throughout the entire compilation pipeline.

Figure 1 illustrates a representative *fBug*: a user-defined `__eq__` is silently ignored with `torch.compile`, causing `!=` to fall back to the default built-in behavior and producing incorrect outputs. This motivates a systematic empirical study of *fBug*s, focusing on their root causes, triggers, and implications for DLC reliability.

3.2 TorchDynamo

We choose TorchDynamo as our research object for three key reasons. First, PyTorch's dominance in both research and industry makes the correctness of TorchDynamo, its default compiler frontend, critically important to the deep learning community. Second, TorchDynamo exemplifies a sophisticated Just-In-Time (JIT) compiler frontend that bridges the gap between dynamic, high-level languages (e.g., Python) and static performance optimization. Finally, its execution model introduces distinct and complex failure modes, and its rapid evolution and numerous issues provide ample research cases.

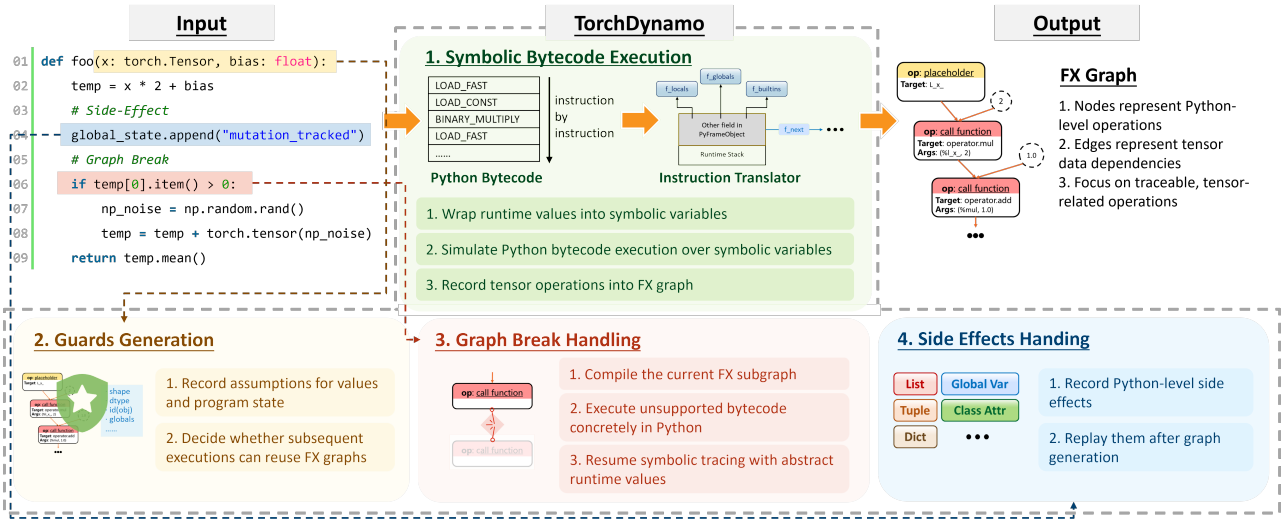

Figure 2: TorchDynamo Workflow

4 Methodology

We attempt to answer the following three research questions.

- **RQ1 (Distribution):** Are *fBug*s prevalent? What are their characteristics in terms of distributions and symptoms?
- **RQ2 (Root cause):** What are the root causes of *fBug*s?
- **RQ3 (Bug detection):** Are these root causes helpful in discovering new bugs?

We design an LLM-aided empirical study methodology. As illustrated in Figure 3, our study involves three stages: (1) collecting high-quality *fBug*s from real-world issue reports and distill DLC frontend specific knowledge, (2) analyzing these bugs by leveraging LLMs' code understanding and summarization capabilities, and (3) generating new test cases with an LLM, guided by our identified root causes, to detect similar bugs in new version.

4.1 Data & Knowledge Preparation

4.1.1 Data preparation. We obtain high-quality experimental data using a two-phase collection process, starting with automatic issue filtering via the GitHub API. The filtering criteria are as follows.

- **Closed issues.** We focus on closed issues, as their resolutions provide definitive outcomes for our analysis.
- **Timeliness.** The issues should be between *2024-06-30* and *2025-06-30*, ensuring that these bugs arose in modern PyTorch 2.x with TorchDynamo.
- **TorchDynamo relevance.** We retain only issues labeled ***module: dynamo*** that constitute reports related to `torch.compile`.
- **Code changes for bug fix.** Each issue must link to at least one pull request (PR) that fixes the bug, enabling precise tracking of code changes and analysis of repair strategies.

Further, we manually review all candidate issues to eliminate potential irrelevant cases or noise. We exclude non-defect cases (e.g., API misuse, expected behaviors, or defects from other components), duplicates identified through similar error signatures and developer discussions, as well as feature requests and performance discussions. This process yields a final dataset of **123** confirmed, unique, and relevant TorchDynamo bugs across PyTorch 2.1 to 2.6, providing a reliable basis for subsequent analysis.

4.1.2 Knowledge preparation. By analyzing the documentation and source code, we gain a comprehensive understanding of TorchDynamo, focusing on its workflow and tasks. We observe that TorchDynamo symbolically models DL program execution instruction by instruction to translate operations into an FX graph. Further, we abstract its workflow, illustrated in Figure 2, which consists of four key tasks.

(1) **Symbolic bytecode execution.** This is a fundamental task in the entire workflow. TorchDynamo uses *VariableTracker*s, symbolic proxies, to wrap runtime variables and record behaviors of various data types [2]. It simulates Python bytecode execution using these proxies instead of concrete values, tracking the frame state, including the stack state and local variables. When encountering a PyTorch operation, TorchDynamo creates an FX node with edges representing data dependencies between operations, incrementally building a graph that represents program execution.

(2) **Guards generation.** TorchDynamo uses *guards* to verify that runtime assumptions made during symbolic execution, such as tensor metadata (e.g., shape, dtype), remain valid, enabling the reuse of cached compilations. These guards are checked before executing a compiled graph, and recompilation is triggered if any guard fails. For instance, as shown in Figure 2, TorchDynamo creates guards (Line 1) to verify that `x` has specific tensor metadata and that `bias` is a float, ensuring the cached graph's validity for the current inputs.

(3) **Graph break handling.** Upon encountering a Python bytecode that cannot be handled, TorchDynamo breaks the graph, compiles the captured subgraph up to that point, and returns to the standard Python interpreter. For example, in Figure 2, the data-dependent control flow at Line 6 causes a graph break because `temp`'s concrete value cannot be known in compilation. This allows TorchDynamo to compile compatible regions incrementally while falling back to eager execution elsewhere.

(4) **Side effects handling.** Side effects are observable updates to program state (globals, attributes, containers, or closures) that FX cannot directly represent. TorchDynamo records these updates during symbolic execution and defers applying them until FX graph execution, materializing the necessary ones in the output bytecode. For example, in Figure 2, Line 4 shows an external object mutation handled as a side effect.

Our analysis reveals that (1) *various data structures*, *guards*, *FX graph*, and *side effects* are critical entities in the compilation pipeline; (2) these four tasks are essential for compilation, but their complexities can introduce bugs.

We structure the analysis and summary into a well-defined format of named tasks with descriptions, creating DLC-frontend-specific knowledge that helps LLMs avoid ambiguity and knowledge conflicts.

4.2 LLM-Aided Bug Analysis

We propose an LLM-human collaboration approach for comprehensive *fBug* analysis, decomposed into two steps: issue report analysis and root cause categorization. This approach leverages LLMs to efficiently understand and summarize bugs, complemented by human expertise for validation.

To extract structured information from unstructured issue reports, each issue is analyzed by an LLM (GPT-5 [43]) and characterized by four fields: *related task*, *trigger*, *symptom*, and *summary*, capturing *when*, *why*, and *what* an issue manifests. This structured representation provides a uniform format suitable for subsequent statistical analysis and categorization.

The *related task* is selected from the four tasks identified in Section 4.1, prompting the LLM to analyze each issue in the context of TorchDynamo's core functionalities. The *symptom* is drawn from three enumerated categories—*crash*, *timeout*, and *inconsistency*—consistent with the classification scheme adopted in prior empirical studies [26, 28, 50]. The *trigger* and *summary* are expressed in natural language. To ensure consistent outputs and reduce variance, we restrict output length and provide a one-shot example.

With these 123 initial analysis results, we cluster bugs by root cause and further divide each cluster by trigger. To ensure reliable results, we query two LLMs (GPT-5 [43] and Claude-Sonnet-4 [3]) and involve two authors to review all LLM-generated annotations. All disagreements are resolved through manual inspection and discussion until consensus is reached.

4.3 LLM-Aided Bug Detection

We employ an LLM (GPT-5 [43]) to generate new test cases that are semantically similar to known bugs but target different situations. Specifically, we design a root cause-aware test case generation strategy. For each identified root cause, we incorporate its name and a description explaining how such bugs occur with respect to the program structure, semantic features, related entities, and functions in an LLM prompt. In addition, we impose several constraints on the generated tests. Each test must be self-contained, as simple as possible, and executable with and without `torch.compile`. We also prompt the LLM to generate corresponding assertions to compare the execution results. Finally, we execute each generated test case in a recent version of PyTorch[1], following the idea of differential testing [41], i.e., comparing results between eager and compile modes. This allows us to detect potential regressions and assess whether semantically similar bugs can still be triggered.

5 Study Result

This section presents our empirical study results and the answers to the research questions. Beyond that, we draw multiple valuable findings and implications.

5.1 Bug Distribution (RQ1)

Table 1: Distribution of *fBug*s

	Symbolic Exec.	Graph Break	Side Effects	Guards	Total
custom class	20	0	6	6	32
container	14	1	0	1	16
iterator	5	0	0	0	5
closure	1	4	2	1	8
Python scalar	4	1	0	3	8
tensor op	4	0	1	3	8
decorator	1	0	1	0	2
global state	0	0	6	1	7
in-place op	0	0	5	0	5
ctx manager	0	11	0	1	12
compile cfg	0	1	0	1	2
unsup. op	8	3	0	0	11
others	0	3	2	2	7
Total	**57**	**24**	**23**	**19**	**123**

tensor op, *in-place op*, and *unsup. op* refer to tensor operations, in-place operations, and unsupported operations, respectively.

Table 1 summarizes the distribution of 123 *fBug*s across the four key tasks and related entities, based on our understanding of TorchDynamo (Section 4.1).

From the task perspective, *symbolic execution* accounts for the majority of defects (57/123), significantly more than *graph break* (24/123), *side effects* (23/123), and *guards* (19/123). This aligns with symbolic execution's complexity and foundational role: as the backbone of graph capture, it is both more error-prone and more frequently exercised, thus accounting for a large share of bugs.

From the entity perspective, *custom classes* (32) and *containers* (16) are the entities most susceptible to errors, primarily during *symbolic execution*, with fewer issues in *guards* and *side effects*. This suggests that TorchDynamo graph capture struggles significantly with dynamic custom class objects (e.g., method resolution and introspection) and containers (e.g., lists and tuples).

The *fBug* distribution reveals strong correlations between tasks and entities. For example, *global state* (6/7) and *in-place op* (5/5) issues are largely confined to *side effects*, concerning the tracking and replaying of side effects. Iterator-related issues appear exclusively in symbolic execution (5/5), reflecting the difficulty of simulating this data structure's execution. *Closure* and *Others* are exceptions, distributed across multiple tasks due to interaction bugs not attributable to a single mechanism.

[1]We use PyTorch 2.10 in this experiment.

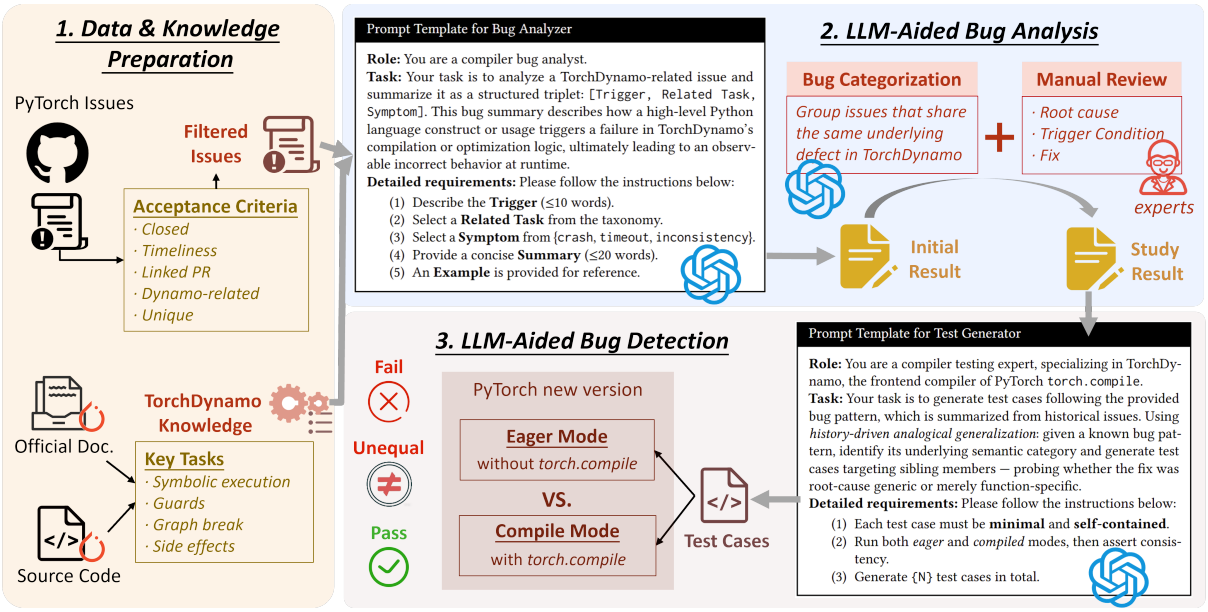

Figure 3: Methodology Overview of Our LLM-Aided Empirical Study

In our analysis of the symptoms of the 123 bugs, *crashes* dominate (88/123), indicating that most reported bugs manifest as immediate execution breakdowns. *Inconsistencies* (28/123), though fewer, are equally critical because silent errors are difficult to detect while having serious consequences. *Timeouts* are rare (7/123), and mostly (6/7) related to guards. Overall, reported TorchDynamo bugs tend to be explicit failures rather than silent errors. This is consistent with expectations, as the former are easier to detect and report.

Finding #1: *fBug*s concentrate in *symbolic execution* (57/123), particularly with high-level Python abstractions such as *custom classes* and *containers*, and exhibit strong correlations between tasks and related entities.

Implication #1: DLC frontends should strengthen the symbolic simulation of Python's custom objects. The entity-task correlations can serve as indicators to guide bug localization.

5.2 Root Cause (RQ2)

Table 2 summarizes our taxonomy of root causes for the 123 *fBug*s, comprising 7 categories and 15 subcategories.

A. Wrong modeling of Python objects. This category accounts for 18.7% (23/123) of the bugs, making it a major root cause in our dataset. It is further divided into four subcategories.

A.1 Incorrect tracing of object initialization. TorchDynamo traces object operations symbolically, which can cause delegated or chained constructor calls to be skipped or only partially traced. This can lead to missing attribute definitions, potentially causing failures or inconsistent behavior during subsequent attribute accesses or mutations. Listing 1 exemplifies PyTorch Issue #145284 [17], where TorchDynamo invokes user-defined methods on `nn.Module` before it is fully initialized. This triggers an `AttributeError` because the symbolic tracer fails to track the internal fields `self.key_cache`.

```
class Cache(torch.nn.Module):
    def __init__(self):
        self.key_cache = []
    def __len__(self):
        return len(self.key_cache)
@torch.compile
def fn(x):
    cache = Cache()
    if cache:   # triggers __len__()
        return x
```

Listing 1: Code Snippet of PyTorch Issue #145284

A.2 Incomplete modeling of attribute access. TorchDynamo parses attributes with fixed symbolic protocols, which may miss edge cases such as dynamic lookups and shadowed callables, resulting in incorrect internal state updates. In Issue #134844 [11] (Listing 2), the function `add_one` is assigned as an attribute to the tensor `t`. However, this is not tracked by TorchDynamo and subsequent invocation of the callable attribute (i.e., `t.add_one(t)`) fails.

```
@torch.compile
def foo(t):
    def add_one(x): ...
    t.add_one = add_one
    return t.add_one(t)
```

Listing 2: Code Snippet of PyTorch Issue #134844

A.3 Overlooked magic method overrides. Python's support for magic methods and operator overloading allows users to customize object behavior, but TorchDynamo may overlook user-defined overrides, assuming default built-in behaviors. In PyTorch Issue #150265 [18] (Listing 3), `3 * x` triggers `__rmul__` on a custom tensor subclass. Due to the wrong logic of `has_torch_function`,

Table 2: Taxonomy of *fBug* Root Causes

Category	Subcategory	Num
A. Wrong modeling of Python objects	A.1 Incorrect tracing of object initialization	3
	A.2 Incomplete modeling of attribute access	8
	A.3 Overlooked magic method overrides	7
	A.4 Misinterpretation of descriptors	4
B. Wrong modeling of containers	B.1 Mis-modeled container abstraction & instantiation	9
	B.2 Untracked container mutation	6
C. Desynchronization of iterator state	–	4
D. Missing type transformation	–	8
E. Wrong handling of execution context & scope	E.1 Uncaptured context managers	7
	E.2 Ignored exception handling	3
	E.3 Wrong closure & variable capturing	5
	E.4 Untracked global state & environment changes	9
F. Uncaptured side effects	F.1 Untracked object attribute updates	9
	F.2 Untracked tensor & metadata updates	7
G. Guard overspecialization & deficiencies	G.1 Overspecialization of mutable or ephemeral state	6
	G.2 Overspecialization of dynamic shapes	5
	G.3 Missing or incomplete guards	7
H. Others	–	16
Total	–	**123**

TorchDynamo misclassifies the multiplication as a generic Python magic-method call rather than a tensor operator, causing an unexpected graph break.

```
class MyParam(torch.nn.Parameter):
    def __new__(cls, data):
        return torch.Tensor._make_subclass(...)
@torch.compile
def f(x):
    return 3 * x   # triggers __rmul__
```

Listing 3: Code Snippet of PyTorch Issue #150265

A.4 Misinterpretation of descriptors. Python descriptors enable dynamic attribute binding by intercepting attribute access at runtime. When TorchDynamo fails to model this protocol correctly, it may simplify descriptor-backed attributes into plain functions or unbound methods based solely on static bytecode patterns. In PyTorch Issue #155841 [21] (Listing 4), the method `incr` is decorated with `functools.lru_cache`. However, TorchDynamo models `f.incr` as a plain function rather than a bound method, thereby failing to implicitly bind `self` to the instance `f`. As a result, the call `f.incr(3)` lacks the instance argument, leading to an argument mismatch and compilation failure.

```
class Foo:
    @functools.lru_cache
    def incr(self, val): ...
@torch.compile
def fn():
    Foo().incr(3)   # descriptor binding lost
```

Listing 4: Code Snippet of PyTorch Issue #155841

Finding #2: A key reason for *fBug*s is the gap between TorchDynamo's symbolic execution and Python's dynamic object model, particularly in *object initialization*, *state mutation*, *dynamic dispatch*, and *runtime binding*.

Implication #2: DLC frontends must ensure full symbolic materialization of objects before they enter the tracing pipeline.

B. Wrong modeling of containers. This category comprises 15 bugs (12.2%), rooted in a fundamental conflict in TorchDynamo: containers are not fully traced, yet their constituent tensors are first-class citizens. This triggers inconsistencies during container-dependent tensor access or control flow. We further divide this category into two subcategories based on container operation types.

B.1 Mis-modeled container abstraction & instantiation. This subcategory covers issues arising from container construction and abstraction, often triggered by automatic entry initialization, reflective access, and dynamic attribute binding (e.g., attaching functions or fields at runtime). In Issue #141118 [13] (Listing 5), TorchDynamo misidentifies `MyWeirdDict`, a class with multiple inheritance, as a pure `nn.Module`. Thus, when accessed as a mapping, the call is erroneously routed through unsupported module-handling logic.

```
class MyWeirdDict(collections.abc.MutableMapping, nn.
    Module):
    def __getitem__(self, item): ...
    def __setitem__(self, key, value): ...
    def __delitem__(self, item): ...
    def __len__(self): ...
    def __iter__(self): ...
@torch.compile
def to_weird_dict(td):
```

```
    return MyWeirdDict(**td)
```

Listing 5: Code Snippet of PyTorch Issue #141118

B.2 Untracked container mutation. This subcategory captures issues caused by container mutations during execution. Representative cases include in-place updates and mutations under dynamic control flow (e.g., data-dependent iteration or exception paths). In Issue #133063 [9] (Listing 6), a generator iterates over `self.mods` to extend list `y` while referencing `y[-1]`. Each iteration is expected to observe the latest state of `y`, but TorchDynamo fails to model this dynamic evolution and reads stale values.

```
@torch.compile
class TempOpModel(nn.Module):
    def __init__(self):
        self.m = nn.ModuleList([...for _ in range(2)])
    def forward(self, x):
        y = [x, x]
        y.extend(m(y[-1]) for m in self.m)
        return y
```

Listing 6: Code Snippet of PyTorch Issue #133063

Finding #3: TorchDynamo's inconsistent tracking of containers and their tensors causes container-related bugs, particularly when containers are dynamically modified, contain non-tensor elements, or are nested.

Implication #3: DLC frontends should adopt a unified, structure-aware tracking strategy that propagates monitoring through container boundaries.

C. Desynchronization of iterator state. This category, containing 4 bugs (3.25%), arises from TorchDynamo's inadequate modeling of iterator states during symbolic execution. Python iterators are highly flexible and stateful, making their behavior difficult to model symbolically. In Issue #128944 [7] (Listing 7), for instance, a `for`-loop consumes a lazy `filter` iterator whose elements are consumed on demand. TorchDynamo fails to accurately capture the internal state transitions of such iterables, causing desynchronization between tracing and execution and triggering unexpected graph breaks.

```
@torch.compile
def fn(inputs):
    it = filter(lambda t: t.requires_grad, inputs)
    out = inputs[0]
    for x in it:   # stateful consumption
        out = out * x
    return out
```

Listing 7: Code Snippet of PyTorch Issue #128944

Finding #4: TorchDynamo's imprecise iterator modeling causes desynchronization between symbolic and concrete execution, especially for lazy iterables and generators with runtime-dependent evaluation.

Implication #4: DLC frontends should explicitly model iterator state, including materializing iterator outputs into trackable symbolic values, modeling iteration semantics, and maintaining state updates across loop boundaries.

```
@torch.compile
class TestModel(torch.nn.Module):
    def forward(self, x, shape_params):
        return torch.ops.aten.view.default(x,
            shape_params)
x = torch.randn(24)
shape_params = [ torch.tensor(2, dtype=torch.int32),
                 torch.tensor(3, dtype=torch.int32),
                 torch.tensor(4, dtype=torch.int32) ]
```

Listing 8: Code Snippet of PyTorch Issue #156720

D. Missing type conversion. This category contains 8 bugs (6.50%), primarily due to inconsistent handling of tensors and Python primitives (e.g., scalar, None, boolean, string). The lack of explicit type normalization results in incorrect comparisons, arithmetic operations, and control-flow predicates. In Issue #156720 [22] (Listing 8), scalars passed as shape parameters within a Python container cause an error because `aten.view.default` expects shape arguments of `SymInt` or `int`; however, the arguments remain unconverted `FakeTensor` objects, triggering an error.

Finding #5: TorchDynamo accurately proxies tensors but neglects crucial type and value conversions for non-tensors, leading to type errors in returns and comparisons.

Implication #5: DLC frontends should consider the types of non-tensor values, not just symbolic tensor proxies.

E. Wrong handling of execution context & scope. This category contains 24 bugs out of 123 (19.5%), the largest share in our dataset. We further divide it into four subcategories.

```
@contextlib.contextmanager
def g(x):
    try:
        yield x.sin()
    finally:
        pass
@torch.compile
def fn(x):
    with g(x) as y:
        z = y + 1
    return z
```

Listing 9: Code Snippet of PyTorch Issue #130559

E.1 Uncaptured context managers. In Python, context managers define runtime execution scopes via the `with` statement. Entering and exiting the `with` block is mandated by Python's context management protocol. However, TorchDynamo may skip unrecognized context managers, incompletely emulate the `with`-block stack, or lose context state across graph breaks. In Issue #130559 9 (Listing 9), the context manager is defined via `@contextlib.contextmanager`, which transforms a generator function into a context management

object. However, this indirect definition of a context manager is not properly recognized and triggers an unexpected graph break.

E.2 Ignored exception handling. Python uses `try/except` blocks to handle runtime errors and provide fallback logic. This subcategory covers issues in which TorchDynamo bypasses or mishandles the exception handler, skipping intended fallback paths. For example, PyTorch Issue #153605 [20] (Listing 10) is triggered by accessing a nonexistent attribute, which raises an `AttributeError`. Although this access occurs inside a `try/except` block, compilation crashes immediately instead of entering the `except` branch.

```
@torch.compile
class Model(nn.Module):
    def forward(self, x):
        try:
            return torch.nn.functional.ATTR
        except AttributeError:
            return x
```

Listing 10: Code Snippet of PyTorch Issue #153605

E.3 Wrong closure & variable capturing. Python functions can access variables from enclosing scopes, such as local variables and nested functions. However, incorrect construction of closure environments during function inlining may lead to errors, particularly when closures interact with containers or global variables. In Issue #136814 [12] (Listing 11), nested function `fn` returns a closure `inner` that captures variable `x`. TorchDynamo is expected to inline both `fn` and `inner`, but incorrectly reconstructs the closure environment, triggering an unexpected `AttributeError` when accessing the captured variable.

```
def test():
    x = torch.ones(1)
    def fn():
        def inner():
            return x + 2
        return inner
    @torch.compile
    def start():
        fn_inner = fn()
        res = fn_inner()
        return res, fn_inner
    start()
```

Listing 11: Code Snippet of PyTorch Issue #136814

E.4 Untracked global state & environment changes. Global variables and runtime configurations are frequently read and modified during program execution. However, to reduce the complexity of tracing, TorchDynamo's symbolic evaluation initially assumes the global state is stable. This tracking strategy can fail to observe or propagate global state mutations, mishandle environment-dependent settings, or improperly apply cached graphs. In Issue #132165 [8] (Listing 12), both functions in file `test_import.py` modify the global variable `global_flag`. In `repro.py`, the compiled function `fn` invokes them sequentially, mutating the same global variable twice. Since both calls originate from the same module object, TorchDynamo redundantly registers it in its side effect tracking system, leading to an assertion failure.

```
# This repro requires two files
# file 1: test_import.py
global_flag = False
def set_flag_true():
    global global_flag
    global_flag = True
def set_flag_false():
    global global_flag
    global_flag = False

# file 2: repro.py
import test_import
@torch.compile()
def fn(x):
    test_import.set_flag_true()
    test_import.set_flag_false()
    return x + 1
```

Listing 12: Code Snippet of PyTorch Issue #132165

Finding #6: Mishandling of execution context is the largest bug category, indicating TorchDynamo's tensor-centric design overlooks context-dependent state changes.

Implication #6: DLC frontends should precisely capture and restore execution contexts to avoid semantic divergence from Python runtime behavior.

F. Uncaptured side effects. This category contains 15 bugs (12.2%) and is further divided into two subcategories.

F.1 Untracked object attribute updates. TorchDynamo may not correctly capture object attribute updates via non-standard patterns, e.g., aliased references or dynamic addition/deletion, leading to potential failures in deferred attribute writes. In Issue #143756 [14] (Listing 13), the constructor of `Something` mutates its attribute storage via a direct write to `self.__dict__`. This direct update to low-level mapping escapes TorchDynamo's side effect tracking, leading to an unexpected graph break during compilation.

```
class Something:
    def __init__(self) -> None:
        self.__dict__["something"] = 'whatever'
@torch.compile
class MyModule(torch.nn.Module):
    def forward(self, x) -> torch.Tensor:
        Something()
        return x
```

Listing 13: Code Snippet of PyTorch Issue #143756

F.2 Untracked tensor & metadata updates. Stateful tensor updates may affect tensor metadata such as shape and dtype, particularly through in-place operations. TorchDynamo does not always anticipate these changes, resulting in incomplete modeling. In Issue #134820 [10] (Listing 14), a custom autograd function performs an in-place update via an `out=` argument. This write is expected to bypass autograd checks, but TorchDynamo inlines the `forward` method without properly modeling the no-grad boundary. Consequently, the `out`-variant observes an incorrect autograd state and raises an error.

```
class toy_fn(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        torch.exp(x, out=x)
        return x
@torch.compile
def f(x):
    return toy_fn.apply(x)
```

Listing 14: Code Snippet of PyTorch Issue #134820

Finding #7: Deferred side effect handling introduces semantic gaps. Inaccurate tracking or collapsing of updates by TorchDynamo can break ordering and state consistency with the original Python program.

Implication #7: DLC frontends should enforce precise ordering guarantees during side effect replay to preserve the original execution semantics.

G. Guard overspecialization & deficiencies. This category contains 18 bugs (14.6%), further divided into three subcategories.

G.1 Overspecialization of mutable or ephemeral state. TorchDynamo might over-specialize short-lived, instance-specific program states, misinterpreting them as stable constants. In PyTorch Issue #128319 [6] (Listing 15), `self.counter+=1` mutates a module attribute during each call. TorchDynamo inserts value-equality guards on this attribute, causing the guards to frequently invalidate and trigger repeated recompilation, which significantly increases compilation overhead.

```
class Mod(torch.nn.Module):
    def __init__(self):
        self.counter = 0
    def forward(self, x):
        self.counter += 1
        return self.linear(x)
opt_mod = torch.compile(Mod())
for _ in range(10):
    opt_mod(torch.randn(1, 1))
```

Listing 15: Code Snippet of PyTorch Issue #128319

G.2 Overspecialization of dynamic shapes. This subcategory covers bugs caused by overspecialization of tensor shapes. TorchDynamo may emit value-equality guards for dimensions that vary across inputs (e.g., batch sizes or mapped axes), effectively treating dynamic dimensions as constants. In Issue #150540 [19] (Listing 16), a scalar tensor is converted to a concrete integer via `x.item()`, and shape constraints are checked using `_check_is_size` and inequality guards. However, the symbolic relation between the derived index `b` and the constraint `b<y.shape[0]` is not preserved during export or compilation, leading to guard validation failures.

```
@torch.compile
class M(torch.nn.Module):
    def forward(self, x, y):
        b = x.item()
        torch._check_is_size(b)
        torch._check(b < y.shape[0])
        return y[:b]
```

Listing 16: Code Snippet of PyTorch Issue #150540

G.3 Absent or insufficient guards. This allows compiled graphs to execute under invalid assumptions, leading to incorrect computations or unexpected exceptions. In PyTorch Issue #132692 [8] (Listing 17), the nested closure `forward_function` captures the module field `self.linear`. During compilation, guard generation depends on a guard manager initialization path. When the flag `enable_cpp_guard_manager` is disabled, the alternative guard path is taken without proper manager setup, leading to assertion failures during guard creation for closure-bound module variables.

Table 3: *fBug*s Detected in Recent Release

(Sub)category	New Issue
A.1	**#176596** , #176692
A.3	**#150765** , **#174050** , **#175841** , **#176679** , **#176686** , #175615, #175943
A.4	**#176599**
C	**#175610** , #175604, #175855, #176693
E.1	#176156
E.2	**#174166** , **#175608** , **#175846**
E.4	**#175857** , **#175953** , #176252
F.1	**#176851**
F.2	**#176854**
Total	**23**

The 15 issues in **gray boxes** are confirmed, the other 8 are pending.

```
class LinearModel(torch.nn.Module):
    def forward(self, x):
        def forward_function(x):
            return self.linear(x)
        return forward_function(x)
torch._dynamo.config.enable_cpp_guard_manager = False
torch.compile(M())(...)
```

Listing 17: Code Snippet of PyTorch Issue #132692

Finding #8: Inappropriate guard designs commonly cause *fBug*s. Overspecialized or missing guards break the balance between correctness and reuse, leading to either excessive recompilation or incorrect graph reuse.

Implication #8: Guard mechanisms should precisely capture necessary assumptions while avoiding unnecessary specialization that harms reuse and performance.

H. Others. Notably, 16 issues are highly specific and too varied to form meaningful groups. For example, in PyTorch Issue #144461 [16], a PyTorch built-in benchmarking utility (i.e., *ThroughputBenchmark*) unexpectedly mutates the global autocast dtype as a side effect of its internal execution, changing it from FP32 to BF16. TorchDynamo's global state guard then detects this mismatch and raises a `RecompileError`, aborting execution. The failure is triggered by an unexpected global state mutation introduced by an external benchmarking utility. Since no other similar APIs exhibit this behavior, it represents an isolated and specific bug.

5.3 Bug Detection (RQ3)

We observe that many bug fixes are case-specific, addressing only the immediate trigger rather than the complete underlying root

cause. Take the Issue #142055 [15] in Figure 1 as an example, PR #142078 [23] (*[dynamo] Properly handle != under user-defined __eq__*) only addresses this particular case where `__eq__` or `__ne__` are overridden. This fix is narrowly scoped and motivates our LLM-aided detection for similar bugs beyond the exact patched condition.

We follow the method in Section 4.3 to generate ten test cases for each (sub)category in Section 5.2. We run all 170 test cases on PyTorch 2.10 and manually analyze each failure or inconsistency encountered. Overall, 23 out of the 170 test cases trigger failures or errors. We have reported these bugs, and 15 have been confirmed, covering 8 (sub)categories. Ten confirmed issues have been resolved with corresponding PRs. Table 3 lists our newly detected bugs.

Notably, five issues are tagged as ***high priority***, indicating that they have a significant impact on DLC stability and deserve urgent attention. Issue #174166 was labeled as ***good first issue***, indicating that it provides insight into a new class of bugs. Issues #174050 and #176679 were tagged as ***module: correctness (silent)***, which is particularly concerning, because these bugs may produce incorrect results without raising any exceptions or warnings.

Furthermore, the newly uncovered *fBug*s reveal a prevalence of case-specific fixes over systematic root cause resolution. For instance, PR #175611 only patched a specific `AttributeError` for #174166, leaving the other two issues (#175608, #175846) with the same root cause (**E.2**) unresolved. This reactive patching, also evident in root cause **A.3**, suggests that systematic remediation remains challenging, especially when root causes stem from Python's dynamic language features.

We attribute the failure to discover new bugs in the remaining categories to four factors: (1) some categories require the interaction of multiple source files; (2) some others depend on specific external API calls from third-party libraries; (3) some are hardware-specific and only manifest in certain GPU environments; (4) and some involve APIs that have since been removed or renamed. These cases fall outside the scope of our current approach or cannot be reproduced in our environment.

Finding #9: TorchDynamo maintenance often relies on fragmented, case-specific patches. Nevertheless, the root-cause-aware test case generation is underscored by the discovery of new *fBug*s in 47% (8/17) of the subcategories.

Implication #9: Bug root causes can effectively guide an LLM to synthesize test cases in detecting new *fBug*s.

6 Discussion

6.1 LLM Aids Empirical Study

LLMs significantly accelerate our study, as evidenced by PyTorch Issue #142055 [15] in Figure 1. While manual analysis, reading reports, running code, and reviewing fixes take approximately one hour, our LLM-based pipeline generates structured summaries in <30 seconds. Combined with <10 minutes of human verification, this approach yields an 80% reduction in analysis time. Beyond efficiency, LLMs improve categorization consistency; guided by domain-knowledge augmented prompts, they reliably map complex issues (e.g., those involving Python magic methods) to potential root causes that human analysts might overlook.

Notably, our methodology is LLM-agnostic, as domain knowledge is explicitly encoded in the prompt. All LLM outputs undergo cross-validation and independent review by two authors, ensuring that the LLM serves as an accelerator rather than a decision-maker and that conclusions remain robust across different models.

6.2 Implications

For DLC developers: Our findings highlight the impedance mismatch between Python's dynamic semantics and static graph requirements. To prevent silent invalidation of tracing, DLC frontends must move beyond ad hoc bug handling toward exhaustive state tracking and robust, semantic-aware guard mechanisms.

For framework users: To ensure reliable acceleration, users should adopt compiler-friendly programming, prioritizing functional purity and explicit state management to mitigate unexpected behavioral divergences during compilation.

For testers: Integrating domain knowledge, such as design assumptions and known bug root causes, with LLMs enables a scalable paradigm for synthesizing effective tests.

6.3 Threats to Validity

Internal threat. To mitigate potential LLM hallucinations and ensure analysis accuracy, we integrate domain knowledge into prompt engineering and manually verify all LLM-generated outputs.

External threat. While our study focuses on a single DLC, the identified root causes characterize graph IR transformations common across mainstream deep learning compilers. Furthermore, our LLM-aided methodology is transferable to other DLC frontends via prompt tuning.

6.4 Limitations and Future Work

First, our test case synthesis is guided by root causes from known *fBug*s; for unseen bugs that fall outside the identified categories, the effectiveness of root-cause-aware generation is a concern. Second, as the DLC ecosystem evolves and more issues are reported, the stability and representativeness of our categorization may shift. Future work would extend the study using larger, more diverse datasets to further broaden the generalizability of our findings.

7 Related Works

The most related work includes testing and empirical studies of DL systems, especially DL frameworks and compilers. We also briefly review LLM-aided empirical study and traditional compiler testing.

Empirical studies on DL system bugs. Research in this domain bifurcates into DL frameworks and compilers. For DL frameworks, existing studies [4, 29, 30, 42, 54, 69, 70] manually categorize bug reports by root cause and symptom. They consistently identify training-phase algorithmic errors as primary, high-effort defects, mainly due to incorrect implementation of DL models and algorithms. Conversely, empirical research on DLCs is relatively sparse. Initial systematic analyses [26, 50] of mainstream DLCs [5, 24, 47, 49, 59] identified tensor-type issues as a major root cause, which led to the development of TVMfuzz [26]. More recently, Huang et al. [28] analyzed false-positive reports in TVM

and OpenVINO [56]. In contrast to these labor-intensive, broad-scope surveys, our study is more focused and can achieve greater analytical depth and scalability.

DL system testing. DL frameworks and DLCs are two main types of objects under test. Existing testing approaches for DL frameworks predominantly leverage fuzzing, ranging from API- and operator-level tests [25, 53, 63, 65], which target API behaviors and individual operator implementations, to model-level tests [27, 34, 37, 46, 61, 66], which synthesize diverse computation graphs to expose inconsistencies across frameworks. DLC testing evolves with the domain knowledge used. Early black-box strategies [38, 44] mutate compiler IRs without internal insights. Subsequent approaches incorporate structural knowledge of computation graphs [60, 73] or operator specifications [51] to guide test generation. More recently, white-box methods such as WhiteFox [67] and OATest [52] utilize LLMs to extract optimization semantics from source code for optimization-aware testing. While progress has been made, current test generation relies on historical data. Our work, however, leverages in-depth analysis of the DLC mechanism to systematically characterize vulnerable compiler components and optimization stages, enabling LLMs to perform targeted testing.

LLM-aided empirical study. Recent research uses LLMs to automate empirical studies [35, 36, 71]. Despite GPT-4's identified limitations in domain-specific reasoning [35], targeted pipelines like AutoEmpirical [71] and LLMCluster [36] successfully automate labor-intensive processes, including fault analysis and bug report clustering. Our work also benefits from these advancements, i.e., significantly reducing manual effort in data preparation and analysis while maintaining high-quality insights.

Traditional compiler testing. Decades of research in conventional compiler testing, including random test case generation [39, 64, 68], optimization-focused testing [32, 33], and formal verification [31, 40], cannot be directly adopted for DLC testing. The unique complexities of DLCs, particularly their tensor-centric computation and domain-specific optimizations, necessitate new testing paradigms beyond traditional approaches.

8 Conclusion

This paper presents the first systematic empirical study of bugs in DLC frontends. Leveraging a domain-knowledge-enhanced LLM, we analyze 123 *fBug*s and derive a bug root cause taxonomy comprising 7 categories and 15 subcategories. Besides, our study provides actionable insights for DLC development and testing. Beyond that, we uncover 23 new *fBug*s (15 confirmed) in later releases, demonstrating its efficacy in guiding DLC testing.

Data Availability

The source code and data are publicly available via Zenodo at https://doi.org/10.5281/zenodo.19229496.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[3] ANTHROPIC. 2025. System Card: Claude Opus 4 & Claude Sonnet 4. https://www-cdn.anthropic.com/6d8a8055020700718b0c49369f60816ba2a7c285.pdf. Accessed: 2026-03-19.

[4] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 135 (Sept. 2023), 31 pages. doi:10.1145/3587155

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[6] PyTorch Contributors. 2024. Issue #128319: [dynamo] Recompilation on a counter-like attribute of nn module. https://github.com/pytorch/pytorch/issues/128319. Accessed: 2026-03-19.

[7] PyTorch Contributors. 2024. Issue #128944: torch.compile graph break due to unsupported builtin filter function. https://github.com/pytorch/pytorch/issues/128944. Accessed: 2026-03-19.

[8] PyTorch Contributors. 2024. Issue #132165: Mutating global variable during inlining a function from imported module breaks in dynamo. https://github.com/pytorch/pytorch/issues/132165. Accessed: 2026-03-19.

[9] PyTorch Contributors. 2024. Issue #133063: torch.compile Parsing error results in error. https://github.com/pytorch/pytorch/issues/133063. Accessed: 2026-03-19.

[10] PyTorch Contributors. 2024. Issue #134820: [Dynamo] propagate required_grad info while applying autograd function. https://github.com/pytorch/pytorch/issues/134820. Accessed: 2026-03-19.

[11] PyTorch Contributors. 2024. Issue #134844: [Dynamo] Handle tensor attributes. https://github.com/pytorch/pytorch/issues/134844. Accessed: 2026-03-19.

[12] PyTorch Contributors. 2024. Issue #136814: Dynamo inlining errors with some calls to nested functions that use captured variables. https://github.com/pytorch/pytorch/issues/136814. Accessed: 2026-03-19.

[13] PyTorch Contributors. 2024. Issue #141118: Dynamo: how to deal with multiple inheritance (nn.Module/MutableMapping). https://github.com/pytorch/pytorch/issues/141118. Accessed: 2026-03-19.

[14] PyTorch Contributors. 2024. Issue #143756: self.__dict__[...] = ... produces a graph break. https://github.com/pytorch/pytorch/issues/143756. Accessed: 2026-03-19.

[15] PyTorch Contributors. 2025. Issue #142055: Dynamo doesn't support != when the compared objects have custom __eq__. https://github.com/pytorch/pytorch/issues/142055. Accessed: 2026-03-19.

[16] PyTorch Contributors. 2025. Issue #144461: ThroughputBenchmark incorrectly change autocast dtype on CPU. https://github.com/pytorch/pytorch/issues/144461. Accessed: 2026-03-19.

[17] PyTorch Contributors. 2025. Issue #145284: [dynamo] torch.compile ICE on using a sourceless unspecialized NN module as branching condition. https://github.com/pytorch/pytorch/issues/145284. Accessed: 2026-03-19.

[18] PyTorch Contributors. 2025. Issue #150265: Graph break on Tensor._make_subclass. https://github.com/pytorch/pytorch/issues/150265. Accessed: 2026-03-19.

[19] PyTorch Contributors. 2025. Issue #150540: PropagateUnbackedSymInts does not know about shape checks in guards. https://github.com/pytorch/pytorch/issues/150540. Accessed: 2026-03-19.

[20] PyTorch Contributors. 2025. Issue #153605: [dynamo] aot_eager can't process try...except when meeting AttributeError. https://github.com/pytorch/pytorch/issues/153605. Accessed: 2026-03-19.

[21] PyTorch Contributors. 2025. Issue #155841: torch.compile fails to trace methods decorated with @lru_cache. https://github.com/pytorch/pytorch/issues/155841. Accessed: 2026-03-19.

[22] PyTorch Contributors. 2025. Issue #156720: torch.compile fails with dynamic shape parameters in view operations. https://github.com/pytorch/pytorch/issues/156720. Accessed: 2026-03-19.

[23] PyTorch Contributors. 2025. Pull Request #142078: [dynamo] Properly handle != under user-defined __eq__. https://github.com/pytorch/pytorch/pull/142078. Accessed: 2026-03-19.

[24] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *CoRR* abs/1801.08058 (2018). arXiv:1801.08058 http://arxiv.org/abs/1801.08058

[25] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational API inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. doi:10.1145/3540250.3549085

[26] Xiaoting Du, Zheng Zheng, Lei Ma, and Jianjun Zhao. 2021. An Empirical Study on Common Bugs in Deep Learning Compilers. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 184–195. doi:10.1109/ISSRE52982.2021.00030

[27] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1418–1430. doi:10.1145/3510003.3510092

[28] Lili Huang, Qingchao Shen, Dong Wang, Yunping Wu, Meng Wang, and Junjie Chen. 2025. False-Positive Bug Reports in Deep Learning Compilers: Stages, Root Causes, and Mitigation. *ACM Trans. Softw. Eng. Methodol.* (Nov. 2025). doi:10.1145/3774889 Just Accepted.

[29] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. doi:10.1145/3377811.3380395

[30] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. doi:10.1145/3338906.3338955

[31] Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo. 2025. Optimization-Directed Compiler Fuzzing for Continuous Translation Validation. *Proc. ACM Program. Lang.* 9, PLDI, Article 172 (June 2025), 24 pages. doi:10.1145/3729275

[32] 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334

[33] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2025. Validating JIT Compilers via Compilation Space Exploration. *ACM Trans. Comput. Syst.* 43, 3, Article 6 (July 2025), 37 pages. doi:10.1145/3715102

[34] Meiziniu Li, Jialun Cao, Yongqiang Tian, Tsz On Li, Ming Wen, and Shing-Chi Cheung. 2023. COMET: Coverage-guided Model Generation For Deep Learning Library Testing. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 127 (July 2023), 34 pages. doi:10.1145/3583566

[35] Jenny T. Liang, Carmen Badea, Christian Bird, Robert DeLine, Denae Ford, Nicole Forsgren, and Thomas Zimmermann. 2024. Can GPT-4 Replicate Empirical Software Engineering Research? *Proc. ACM Softw. Eng.* 1, FSE, Article 60 (July 2024), 24 pages. doi:10.1145/3660767

[36] Yuchen Ling, Shengcheng Yu, Chunrong Fang, Quan Zhou, and Zhenyu Chen. 2025. LLM-based Crowdsourced Test Report Clustering. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2025). doi:10.1145/3765756 Just Accepted.

[37] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. doi:10.1145/3575693.3575707

[38] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (April 2022), 26 pages. doi:10.1145/3527317

[39] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. doi:10.1145/3428264

[40] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. doi:10.1145/3453483.3454030

[41] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10 (1998), 100–107. https://api.semanticscholar.org/CorpusID:14018070

[42] Mohammad Mehdi Morovati, Amin Nikanjam, Florian Tambon, Foutse Khomh, and Zhen Ming (Jack) Jiang. 2024. Bug characterization in machine learning-based systems. *Empir. Softw. Eng.* 29, 1 (2024), 14. doi:10.1007/S10664-023-10400-0

[43] OpenAI. 2025. GPT-5 System Card. arXiv:2601.03267 [cs.CL] https://arxiv.org/abs/2601.03267

[44] David Pankratz. 2020. TVMFuzz: Fuzzing Tensor-level Intermediate Representation in TVM. https://github.com/dpankratz/TVMFuzz. Accessed: 2026-03-19.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library.* Curran Associates Inc., Red Hook, NY, USA.

[46] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. doi:10.1109/ICSE.2019.00107

[47] plaidML Team. 2025. plaidML: A platform for making deep learning work everywhere. https://github.com/plaidml/plaidml. Accessed: 2025-12-31.

[48] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems* 4 (2022), 638–651.

[49] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[50] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. doi:10.1145/3468264.3468591

[51] Qingchao Shen, Yongqiang Tian, Haoyang Ma, Junjie Chen, Lili Huang, Ruifeng Fu, Shing-Chi Cheung, and Zan Wang. 2025. A Tale of Two DL Cities: When Library Tests Meet Compiler. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 305–316. doi:10.1109/ICSE55347.2025.00025

[52] Qingchao Shen, Zan Wang, Haoyang Ma, Yongqiang Tian, Lili Huang, Zibo Xiao, Junjie Chen, and Shing-Chi Cheung. 2025. Optimization-Aware Test Generation for Deep Learning Compilers. arXiv:2511.18918 [cs.SE] https://arxiv.org/abs/2511.18918

[53] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. 2023. ACETest: Automated Constraint Extraction for Testing Deep Learning Operators. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 690–702. doi:10.1145/3597926.3598088

[54] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2023. Silent bugs in deep learning frameworks: an empirical study of Keras and TensorFlow. *Empirical Softw. Engg.* 29, 1 (Nov. 2023), 34 pages. doi:10.1007/s10664-023-10389-6

[55] NVIDIA Team. 2023. Fuser: A Fusion Code Generator for NVIDIA GPUs (commonly known as nvFuser). https://github.com/NVIDIA/Fuser

[56] OpenVINO Team. 2025. OpenVINO: Open-source software toolkit for optimizing and deploying deep learning models. https://github.com/openvinotoolkit/openvino. Accessed: 2025-12-31.

[57] XLA Team. 2017. XLA (Accelerated Linear Algebra). https://openxla.org/xla TensorFlow Dev Summit.

[58] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[59] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[60] Zihan Wang, Pengbo Nie, Xinyuan Miao, Yuting Chen, Chengcheng Wan, Lei Bu, and Jianjun Zhao. 2023. GenCoG: A DSL-Based Approach to Generating Computation Graphs for TVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 904–916. doi:10.1145/3597926.3598105

[61] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium*

on the Foundations of Software Engineering (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. doi:10.1145/3368089.3409761

[62] AI Is Crazy Website. 2023. Pytorch vs Tensorflow: A Head-to-Head Comparison. https://aiiscrazy.com/pytorch-vs-tensorflow-a-head-to-head-comparison/. Accessed: 2025-06-28.

[63] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 995–1007. doi:10.1145/3510003.3510041

[64] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. doi:10.1145/3597503.3639121

[65] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 176–188. doi:10.1145/3533767.3534220

[66] Xiaoyuan Xie, Yan Song, Songqiang Chen, and Jinfu Chen. 2026. Subgraph-Oriented Testing for Deep Learning Libraries. *IEEE Transactions on Software Engineering* 52, 3 (2026), 908–922. doi:10.1109/TSE.2026.3655712

[67] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 296 (Oct. 2024), 27 pages. doi:10.1145/3689736

[68] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[69] Sharon Chee Yin Ho, Vahid Majdinasab, Mohayeminul Islam, Diego Elias Costa, Emad Shihab, Foutse Khomh, Sarah Nadi, and Muhammad Raza. 2023. An Empirical Study on Bugs Inside PyTorch: A Replication Study. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 220–231. doi:10.1109/ICSME58846.2023.00031

[70] Brian Yu, Rubayet Rahman Rongon, Chen Cao, and Xuechen Zhang. 2024. A Study of PyTorch Bug Patterns and Memory-Related Challenges. In *2024 IEEE International Conference on Big Data (BigData)*. 7586–7591. doi:10.1109/BigData62323.2024.10824945

[71] Jiongchi Yu, Weipeng Jiang, Xiaoyu Zhang, Qiang Hu, Xiaofei Xie, and Chao Shen. 2025. AutoEmpirical: LLM-Based Automated Research for Empirical Software Fault Analysis. arXiv:2510.04997 [cs.SE] https://arxiv.org/abs/2510.04997

[72] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. doi:10.1145/3453483.3454106

[73] Chijin Zhou, Bingzhou Qian, Gwihwan Go, Quan Zhang, Shanshan Li, and Yu Jiang. 2024. PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 317 (Oct. 2024), 27 pages. doi:10.1145/3689757