

ShapeTester: Constraint-Aware Two-Stage Test Case Generation for Detecting Shape Errors in Deep Learning Frameworks

Shuo Sun^{1,2}, Yixuan Fu^{1,2}, Shixin Zhang², Jiwei Yan²,
Jun Yan^{2,3*}, Jian Zhang³

¹University of Chinese Academy of Sciences, Beijing, China.

²Technology Center of Software Engineering, Institute of Software,
Chinese Academy of Sciences, Beijing, China.

³Key Laboratory of Systems Software, Institute of Software, Chinese
Academy of Sciences, Beijing, China.

*Corresponding author(s). E-mail(s): yanjun@ios.ac.cn;

Contributing authors: sunshuo20@otcaix.iscas.ac.cn;

fuyixuan21@mails.ucas.ac.cn ; zhangshixin@otcaix.iscas.ac.cn;

yanjiwei@otcaix.iscas.ac.cn; zj@ios.ac.cn;

Abstract

Deep learning (DL) frameworks serve as the foundational infrastructure of the modern artificial intelligence ecosystem, providing essential APIs for the construction, training, and deployment of diverse DL models. Within these frameworks, tensor shape transformations are fundamental to core calculation logic. However, the computational logic underlying these shape transformations is often complex and error-prone. Such shape errors can lead to silent incorrect outputs and consequently trigger severe runtime exceptions. Existing API-level testing techniques primarily target crash-inducing inputs or numerical instabilities, failing to provide specialized generation strategies or oracles for shape-related logic. To bridge this gap, we propose **ShapeTester**, a shape-error-targeted testing approach. ShapeTester employs a two-stage strategy that combines combinatorial testing and genetic algorithms to generate test inputs, exploring the output shape space. To detect shape errors, ShapeTester implements a differential oracle by exploiting the semantic equivalence between static and dynamic execution modes, a fundamental property inherent in modern deep learning frameworks. ShapeTester has successfully uncovered 120 previously unknown errors in mainstream DL frameworks, including TensorFlow, PyTorch, and Keras,

95.8%(115/120) of which have been confirmed by the framework developers. Furthermore, in a controlled evaluation using a benchmark of 20 known shape errors, ShapeTester achieved a 70% detection rate, compared with 50% for the strongest evaluated baselines.

Keywords: Deep Learning Library, API Testing, Shape Error

1 Introduction

Deep learning (DL) frameworks, such as TensorFlow, PyTorch, and Keras, have become fundamental to modern artificial intelligence, empowering developers to efficiently build and deploy complex models across diverse domains. These frameworks provide application programming interfaces (APIs) that abstract low-level computational details, enabling a focus on high-level model design. The correctness and reliability of these DL APIs are crucial, as any defects can propagate through the entire software stack, compromising the accuracy and robustness of the resulting models. Among the various types of API defects, shape errors, i.e., errors occurring inside the APIs when calculating tensor shapes, are particularly prevalent and insidious. According to a recent empirical study [1], shape errors are among the top three most common API error types in deep learning frameworks, accounting for 12.2% of all collected errors. Such internal miscalculations can lead to runtime exceptions or silent incorrect computations, wasting significant development effort and potentially causing critical issues in safety-sensitive applications. Given the fundamental role these APIs play in the AI software stack, systematically testing and ensuring their resilience against shape errors is an important research problem, motivating the need for testing methodologies targeted to shape errors.

In recent years, increasing research efforts have been dedicated to testing DL frameworks. One line of research [2, 3] adopts *model-level testing*, which generates diverse DL models to exercise framework code and expose potential bugs in DL frameworks. Test cases generated by these approaches are close to the DL framework usage in real-world application code. However, as a high-level testing approach, model-level testing often fails to thoroughly explore the parameter space of each layer API. Another line of works [4, 5] adopts *API-level testing*, which generates diverse parameter values to test DL APIs directly. Compared to model-level testing, API-level testing concerns the comprehensive behaviors of each specific API, which can cover the corner cases more efficiently than model-level testing and detect many deep hidden bugs. However, existing API-level testing techniques target crash bugs and numerical errors, leaving shape errors largely unexplored. Consequently, these methods lack specialized test oracles for identifying subtle dimension-calculation errors that do not trigger immediate crashes. Moreover, without shape-aware generation strategies, state-of-the-art (SOTA) techniques often produce test inputs that fail to reach the complex internal logic responsible for tensor transformations. As demonstrated in our preliminary study (Section 4.3), test inputs generated by SOTA testing methods triggered only 10 out of

the 20 shape errors in our benchmark, which underscores a persistent gap in ensuring the reliability of DL frameworks.

To fill this gap, we propose a **shape-error-targeted** testing approach for DL APIs. According to our investigation, two fundamental challenges must be addressed to realize an effective and automated shape-testing methodology.

Challenge 1: Exploring Constrained Shape-Related Parameter Spaces

As DL APIs are widely used, most hidden bugs cannot be triggered straightforwardly, i.e., these bugs may reside in corner cases involving specific parameter combinations. To expose in-depth shape errors, a strategy is needed to systematically explore the combination space of parameters that dictate tensor transformations (e.g., **kernel sizes**, **strides**, and **paddings**). Moreover, DL APIs involve complex constraints among their parameters, and any violation of these constraints leads to an immediate execution failure before reaching the core shape-calculation logic. Consequently, the primary challenge lies in generating test inputs and prioritizing **valid yet diverse** parameter combinations to maximize the exploration of the API’s shape-handling logic.

Challenge 2. Establishing a Test Oracle for Shape Inconsistencies. Existing API-level testing approaches primarily rely on crash detection or cross-backend (e.g., CPU vs. GPU) output comparison as test oracles. While these oracles are effective for uncovering crash bugs and numerical inaccuracies, they are not suited for identifying shape errors. Shape errors, which stem from incorrect tensor dimension calculations inside an API, rarely lead to immediate runtime exceptions; instead, they silently propagate erroneous shapes to subsequent operations, potentially causing degraded model performance or incorrect outputs far downstream. Furthermore, shape errors are typically algorithm-intrinsic and device-independent, meaning they manifest consistently across different hardware backends. Therefore, a specialized oracle is required to capture discrepancies between the actual output shape and the intended mathematical semantics of the API.

Our Approach. To address these challenges, we propose *ShapeTester*, a novel shape-error-targeted testing approach for DL APIs. To address **Challenge 1**, *ShapeTester* employs a two-staged and constraint-aware strategy for test input generation. During the first stage, *ShapeTester* focuses on **Shape-Related Parameters (SRPs)**, i.e., parameters that directly influence tensor shape transformations (e.g., **stride**, **padding**, and **dilation**), and systematically explores the pairwise interactions among them. Specifically, *ShapeTester* constructs a discrete search space via Large Language Models (LLMs). Building upon this foundation, it generates test inputs through pairwise interaction to systematically explore diverse shape-transformation scenarios. To further expose deep bugs beyond pairwise interaction, *ShapeTester* transitions to a shape-diversity guided evolutionary search stage, utilizing the valid test inputs generated in the first stage as the seed population. This search process is driven by a specialized fitness function designed to maximize output shape diversity, thereby incentivizing the exploration of unprecedented tensor transformation behaviors. Moreover, *ShapeTester* incorporates a feedback loop to handle the complex constraints among parameters. When a test input triggers an execution crash, *ShapeTester* analyzes the error message to pinpoint the responsible parameters. If the

crash originates from shape-irrelevant parameters, *ShapeTester* regenerates the test case to ensure the targeted SRP combination is fully exercised. Conversely, if the crash stems from fundamental constraints among SRPs, *ShapeTester* treats this as a discovered constraint and dynamically prunes the combinatorial search space to prevent redundant exploration of invalid regions. To address **Challenge 2**, we design a novel differential oracle to detect shape inconsistencies. Modern DL frameworks typically support two execution modes: dynamic graphs and static graphs. Dynamic graphs perform immediate computation on input tensors, while static graphs rely on symbolic shape inference during a preliminary compilation phase. *ShapeTester* detects shape errors by comparing the output shapes generated in the dynamic mode against the symbolic shapes inferred in static mode. Any discrepancy between these two execution paths reveals a potential shape error in the framework’s shape calculation logic.

We evaluated *ShapeTester* on mainstream DL frameworks, including TensorFlow, PyTorch, and Keras. *ShapeTester* successfully uncovered 120 previously unknown errors, 95.8%(115/120) of which have been confirmed by the framework developers. To assess the quality of test input generation, we compared *ShapeTester* against state-of-the-art baselines on a benchmark of 20 representative APIs. For a fair comparison, we equipped the baselines with the same differential oracle. The results show that *ShapeTester* achieved the highest observed values among the evaluated approaches across three dimensions: bug detection rate (0.70), output shape diversity (0.74), and valid input ratio (0.80).

In summary, the primary contributions of this paper are as follows.

1. **A Constraint-Aware Shape-Testing Methodology.** We propose a novel testing strategy that systematically explores the shape-related parameter space of DL APIs. This methodology integrates combinatorial testing with evolutionary search. It specifically employs a feedback loop to bypass early execution failures caused by complex parameter constraints, ensuring a deep exploration of shape-calculation logic.
2. **A Novel Differential Shape Oracle.** We introduce a specialized test oracle designed to detect silent shape inconsistencies. By comparing the tensor dimensions generated during dynamic graph execution against the symbolic shapes inferred in static mode, our approach identifies internal framework errors that traditional crash-based oracles cannot detect.
3. **Large-scale Evaluation and Bug Discovery.** We implement our approach in an automated tool and evaluate it on three major DL frameworks: TensorFlow, PyTorch, and Keras. Our tool successfully uncovered 120 previously unknown shape-related bugs. To date, 95.8%(115/120) of these bugs have been officially confirmed or fixed by the framework developers.

2 Background and Motivating Example

In this section, we provide the technical background and motivating example for our work. We first introduce the core concepts of Shape-Related Parameters (SRPs) and the dynamic and static execution modes in modern DL frameworks, which form the basis of our testing targets and oracle design. Building on these foundations, we

present a real-world shape inconsistency bug discovered by *ShapeTester* to illustrate the challenges in shape error detection.

2.1 Shape-Related Parameters

In DL frameworks, DL APIs (e.g., `Conv2d`, `Reshape`, `Pooling`) are essentially transformation functions that map one or more input tensors to an output tensor. While a single API may involve numerous parameters, they do not contribute equally to the properties of the resulting output. We categorize these parameters into two distinct groups based on their influence on the geometry of the output tensor.

Shape-Irrelevant Parameters (SIPs): These parameters control the numerical precision, execution device, or metadata of the operation **without** altering the dimensions of the output. Examples include `dtype` (e.g., `float32` vs. `int64`), `device` (e.g., `CPU` vs. `GPU`), and `name`.

Shape-Related Parameters (SRPs): We define SRPs as the subset of API parameters that directly dictate the mathematical calculation of the output tensor shape.

2.2 Dynamic and Static Execution Modes

Modern deep learning frameworks provide two distinct execution modes to balance development flexibility with deployment efficiency: dynamic execution and static execution.

Dynamic Execution (Eager Mode). In dynamic mode, operations are executed imperatively using concrete input tensors. When an API is invoked, the framework immediately dispatches the operation to the underlying compute kernels. The output tensor is materialized in memory, and its dimensions are determined by the actual execution result. The resulting shape is therefore a concrete attribute of the returned tensor object.

Static Execution (Graph or Symbolic Mode). In static mode, the framework constructs or compiles a computational representation of the tested operation. During graph construction, tracing, or compilation, the framework propagates input-shape information and determines the expected dimensions of intermediate and output tensors. This shape-computation path supports graph validation and optimization before or during the execution of the compiled representation.

Although the concrete interfaces differ across frameworks, the dynamic and static paths are intended to implement the same API-level tensor transformation. For the same generated test case, *ShapeTester* therefore keeps the API wrapper and parameter configuration unchanged, obtains a concrete runtime shape through the dynamic path, and obtains the corresponding graph or symbolic shape through the static path. Section 3.4 describes how these two shapes are obtained and compared for TensorFlow, Keras, and PyTorch.

2.3 Motivating Example

We illustrate our motivation utilizing a real-world defect discovered in the `keras.layers.Discretization` API. As shown in `Test 1` of Figure 1, this

Test 1: A true-positive test

```
layer = Discretization(bin_boundaries=[1.0, 2.0, 3.0, 4.0],output_mode='count',
                      num_bins=None, epsilon=1e-6, dtype=None)
layer(np.ones((2, 3, 4))).shape # (2, 3, 5)
layer(keras.Input((3, 4))).shape # expceted:(None, 3, 5) Actual:(None, 3, 4)
```

Test 2: An illegal test

```
layer = Discretization(bin_boundaries=[1.0, 2.0, 3.0, 4.0],output_mode='count',
                      num_bins=5, epsilon=1e-6, dtype=None)
#ValueError: Both `num_bins` and `bin_boundaries` should not be set. Received:
`num_bins=5` and `bin_boundaries=[1.0, 2.0, 3.0, 4.0]`
```

Test 3: A passing test

```
layer = Discretization(bin_boundaries=[1.0, 2.0, 3.0],output_mode='int',
                      num_bins=None, epsilon=1e-6, dtype=None)
layer(np.ones((2, 3, 4))).shape # (2, 3, 4)
layer(keras.Input((3, 4))).shape # expceted:(None, 3, 4) Actual:(None, 3, 4)
```

Fig. 1: A motivating example of a shape inconsistency bug in the `keras.layers.Discretization` API

API exhibits a critical shape inconsistency. When the API is configured with `output_mode='count'` and four `bin_boundaries`, these boundaries partition the value range into five bins; hence, the last output dimension should be five. The dynamic execution path constructs this representation and correctly produces an output tensor of shape `(2, 3, 5)`, whereas the symbolic path reports `(None, 3, 4)`. As documented in Keras issue #22403,¹ for the affected aggregation modes (`count` and `multi_hot`), the symbolic output-shape computation incorrectly preserves an input-dependent dimension regardless of the configured number of bin boundaries, instead of deriving the output dimension from the number of bins. Consequently, the static path omits the fifth bin in this example. The defect therefore lies in the symbolic output-shape computation rather than in the numerical discretization performed during dynamic execution.

When applying SOTA testing approaches (*DeepREL* [6], *NablaFuzz* [7], *TitanFuzz* [8], and *MirrorFuzz* [9]) to this API, all of the approaches failed to expose this bug after generating 100 test inputs. This failure reveals three fundamental limitations of existing approaches: (1) the lack of semantic oracles for validating shape correctness, (2) the inability to handle complex inter-parameter constraints, and (3) the absence of targeted exploration of shape-influencing parameter spaces.

(1) Specialized Shape Oracle. As illustrated by Test 1, shape errors often do not trigger immediate system crashes; instead, they result in the silent generation of tensors with incorrect dimensions. In this instance, the static graph mode fails to represent the fifth bin identified above, reporting `(None, 3, 4)` instead of the expected `(None, 3, 5)`. Consequently, traditional crash oracles are incapable of detecting such defects. Moreover, because shape inference logic is typically algorithmic and device-agnostic, cross-backend differential testing (e.g., comparing CPU vs.

¹<https://github.com/keras-team/keras/issues/22403>

GPU outputs) also fails to expose these inconsistencies, as the incorrect symbolic logic remains identical across hardware targets.

To address this, *ShapeTester* employs a specialized differential oracle that compares the output shapes of the same API across dynamic and static execution modes. As shown in Test 1, this comparison effectively captures the discrepancy between the dynamic runtime shape and the static symbolic shape, revealing the internal logic error.

(2) Complex Parameter Constraints Even with an appropriate oracle, SOTA approaches still failed to trigger the target defect because they do not account for the constraints between API parameters. Deep learning frameworks impose strict preconditions on parameter combinations. For instance, in the `keras.layers.Discretization` API, the `bin_boundaries` and `num_bins` parameters are mutually exclusive. As shown in Test 2, assigning values to both parameters triggers a `ValueError`, which terminates execution before reaching the shape-calculation logic. Extracting such constraints is difficult due to the dynamic nature of Python and the lack of formal specifications in documentation. Consequently, existing fuzzers often generate illegal inputs that result in early exits. While Test 2 contains the parameter values necessary to trigger the bug, the violation of these constraints renders the test case ineffective.

To overcome this, *ShapeTester* incorporates a runtime crash monitoring module that intercepts exception messages during execution. By analyzing these error messages, *ShapeTester* identifies the specific parameter combinations responsible for the conflict and dynamically adjusts subsequent test generations, ensuring the exploration bypasses shallow validation checks to reach deep-seated shape-related logic.

(3) Targeted Exploration of Parameter Combinations. Another reason existing methods fail to trigger the bug is that they lack a targeted strategy for exploring shape-influencing parameter spaces. Current methodologies rely on different guidance mechanisms. For instance, *DeepREL* applies random mutation operators to existing valid seeds, while *MirrorFuzz* leverages historical error patterns and generates constraint-violating inputs. However, to effectively expose shape errors, it is critical to prioritize parameters that directly dictate tensor transformations. As demonstrated in Test 1, the bug is triggered by a specific interaction between `bin_boundaries`, `output_mode`, and the input tensor’s shape. Existing methods spend computational resources on shape-irrelevant parameters, such as `epsilon` or `dtype`. While exploring these values may uncover other types of defects, it contributes little to the detection of shape errors. As shown in Test 3, even when an approach generates a legally executable test input, it fails to trigger the bug if it does not cover the specific combination of shape-influencing parameters required to expose the inconsistency.

To overcome this limitation, *ShapeTester* employs a two-stage test generation strategy. In the first stage, it utilizes pairwise testing to systematically cover interaction scenarios among SRPs. In the second stage, it applies shape-diversity guided evolutionary search to further explore the transformation logic. In our example, the bug in Test 1 was successfully captured during the first stage by specifically targeting the combination of the `bin_boundaries` and `output_mode` parameters.

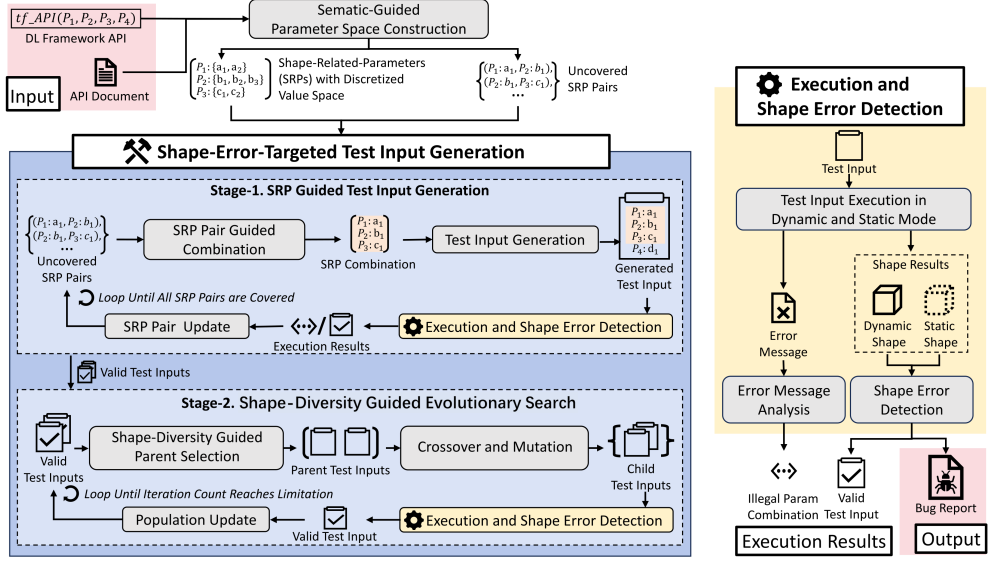


Fig. 2: Method Overview

3 Approach

This section presents the design of *ShapeTester*, our shape-error-targeted testing approach. We first provide an overview of *ShapeTester* and then detail its core components.

3.1 Approach Overview

Fig. 2 illustrates the overall workflow of *ShapeTester*. At its core, *ShapeTester* considers shape testing as a constrained exploration problem over SRP spaces, guided by both structural coverage and behavioral diversity. The workflow is designed as a feedback-driven loop that systematically explores SRP spaces. Specifically, *ShapeTester* first extracts SRPs and their value spaces from API documentation. Then, *ShapeTester* iteratively generates test inputs and refines the parameter space according to the execution result. The test input generation process consists of two complementary stages. The first ensures systematic coverage of fundamental parameter interactions, while the second drives exploration toward rare and previously unseen shape transformations.

Stage-1: SRP Guided Generation. This stage, detailed in Section 3.3.1, aims for broad coverage of SRP interactions. Guided by the principle of pair-wise testing, *ShapeTester* selects uncovered SRP pairs and constructs test inputs that satisfy pairwise coverage. This ensures that the fundamental interactions between SRPs are systematically exercised.

Stage-2: Shape-Diversity Guided Evolutionary Search. This stage, detailed in Section 3.3.2, focuses on corner cases beyond initial combinations. Taking the valid

test inputs from Stage 1 as the initial population, *ShapeTester* performs a genetic evolutionary search. It utilizes selection, crossover, and mutation operators guided by an **output shape diversity** metric that encourages exploration of structurally distinct output-shape profiles. We therefore characterize this stage specifically as evolutionary search.

The generated test inputs are executed in both dynamic execution mode and static execution mode, obtaining concrete runtime shapes and inferred symbolic shapes, respectively. By comparing these results, any discrepancy directly indicates a violation of semantic consistency, revealing a potential defect in the framework’s shape computation logic. Moreover, execution feedback is used to infer parameter constraints and guide subsequent input generation toward valid and bug-prone regions. Constraint-violating executions provide feedback guidance that is used to refine the search space and guide subsequent input generation.

3.2 Semantic-Guided Search Space Construction

The first phase of *ShapeTester* is to define the search space for test generation. This is challenging due to the flexible types of parameters and complex constraints among them. To address this, *ShapeTester* employs a semantic extraction procedure that leverages LLMs to interpret API descriptions and construct a structured parameter space. As highlighted in recent studies [8], LLMs are trained on vast repositories of code and documentation, providing them with prior knowledge of popular DL frameworks. Such general framework knowledge enables the LLM to interpret the documented functionality of an API and the semantic roles of its parameters. This process can be viewed as two key steps: (1) identifying SRPs, and (2) constructing a representative value space for systematic exploration.

We treat SRP identification as a context-dependent semantic classification rather than a fixed classification based solely on parameter names. Given the API documentation and the API code, the LLM examines the parameters and identifies whether changing each parameter can affect the rank or any dimension of the output tensor under at least one supported configuration. A parameter may therefore be classified as shape related even if it affects the output shape only for particular input shapes, parameter values, or execution branches. Conversely, parameters for which no shape influence is identified from the available context are excluded from the SRP search space. The resulting classification determines which parameter domains and interactions are prioritized during subsequent combinatorial and evolutionary exploration.

Accordingly, the SIP/SRP distinction is operational and relative to the analyzed API context and candidate value space. An apparently shape-irrelevant parameter may still have an undocumented or indirect influence on shape, whereas an SRP may affect shape only in a subset of contexts. *ShapeTester* uses the extracted classification to focus its testing budget, but does not assume that the classification constitutes a complete formal specification of all shape dependencies.

Building upon this identification, the LLM constructs a discretized value space for each SRP using the API documentation and source code provided in the prompt. For

Role

You are a deep learning engineer proficient in the {frame_name} framework.

Context

The documentation description of {api_name} is as follows: {api_doc}.
The code of {api_name} is: {api_code}.

Task

1. Construct the value space for each parameter.
 - For discrete parameters, directly list all possible values.
 - For continuous parameters, discretize their value space to ensure that boundary values and at least 5 typical values are included. These typical values should cover all legal value scenarios.
2. Identify all parameters that can affect the shape of the output tensor.

Output Format

```
shape_related_params = ['param-1', ...]
class ValueSpace:
    param1 = [value-1, value-2, ...]
    ...
```

Fig. 3: Prompt template and expected output format for SRP identification and search-space construction.

parameters with a finite discrete domain, the prompt instructs the LLM to enumerate all possible legal values. For parameters with a continuous or prohibitively large numeric domain, the prompt requires the resulting value space to include boundary values and at least five typical values. Here, “typical values” refer to legal, non-boundary values selected by the LLM according to the parameter semantics and implementation to cover different valid usage scenarios, rather than values generated using a fixed numerical sampling rule. By explicitly requiring both boundary values and typical values that cover legal scenarios, the prompt balances the exploration of corner cases with that of ordinary valid configurations. The resulting discretization provides a finite starting space for the subsequent combinatorial and evolutionary stages, rather than exhaustive coverage of the original numeric domain. These discretized spaces are then combined to derive interaction targets among SRPs, which serve as the coverage objectives for systematic exploration.

As shown in Figure 3, the prompt assigns the LLM the role of a deep-learning-framework engineer and supplies the API documentation and source code as context. It requests two structured outputs: a list of parameters that can affect the output tensor shape and a `ValueSpace` class containing their candidate values. The prescribed Python format enables *ShapeTester* to extract the identified SRPs and their value spaces automatically.

This process enables *ShapeTester* to construct a search space that is both expressive of shape semantics and amenable to systematic exploration, which serves as the foundation for the subsequent exploration.

3.3 Shape-Error-Targeted Test Input Generation

As stated in previous sections, shape errors often arise from intricate interactions among multiple parameters, rather than from isolated parameter values. To systematically explore such interactions, *ShapeTester* adopts a two-stage test input generation

strategy. The first stage aims to achieve comprehensive coverage of pairwise combinations among SRPs, ensuring that fundamental parameter interactions are exercised. While pairwise coverage captures fundamental interactions, higher-order interactions often remain unexplored due to combinatorial explosion. To address that, the second stage leverages an evolutionary algorithm to discover more complex interactions beyond pairwise combinations, guided by output shape diversity.

3.3.1 SRP Guided Test Input Generation

This stage considers the SRP interaction exploration as a constraint-aware combinatorial testing problem. Conventional combinatorial testing [10, 11] aims to minimize the test suite size while achieving full pairwise coverage. However, this minimization fails to account for implicit and dynamically enforced parameter constraints. In practice, violations of these constraints often trigger premature execution termination, which prevents the test from reaching the core shape-calculation logic. Consequently, a test input that causes an early crash cannot be considered to successfully cover the targeted SRP interaction.

To address this, *ShapeTester* employs a constraint-aware combinatorial exploration strategy, where coverage is defined over valid executions rather than syntactic parameter combinations. Given k SRPs, where the i -th SRP has a discretized value domain of size d_i , *ShapeTester* constructs $\sum_{1 \leq i < j \leq k} d_i d_j$ pairwise value combinations as coverage targets. In each iteration, the remaining complete configurations are scored according to the uncovered pairwise combinations they cover and the accumulated failure cost associated with their parameter values. The configuration with the highest score is selected to construct the next test input. For the selected configuration, *ShapeTester* then attempts to construct a concrete test input. The framework first queries the pool of previously executed valid test inputs to identify a seed input whose SIP values are compatible with the target SRP combination, where the compatibility is determined by ensuring the seed’s SIP values do not conflict with any constraints recorded in prior iterations. If no such valid input is available, the system leverages an LLM to generate a valid test input that is compatible with the target SRP values.

The size of the resulting search space varies across APIs. APIs with more SRPs or larger discretized value domains contain more pairwise interaction targets and complete input configurations. While the number of pairwise targets is $\sum_{1 \leq i < j \leq k} d_i d_j$, the number of complete configurations is $\prod_{i=1}^k d_i$. Consequently, during the SRP-guided stage, APIs with larger parameter spaces generally require more test inputs to cover their pairwise SRP interactions.

Figure 4 shows the prompt template and expected output formats for test-case completion. The prompt includes the API documentation, wrapper code, current parameter value spaces, and the target SRP values. The LLM first determines whether the specified values conflict. It then returns either a complete test case that preserves the target values or a `conflict_params` list identifying the conflicting parameters, both in the prescribed Python format.

The completed test input is then evaluated by the Execution and Shape Error Detection module. An SRP pair is marked as covered only when the resulting test input executes successfully. When an execution fails, *ShapeTester* analyzes the error

Role You are a deep learning engineer proficient in the {frame_name} framework.	
Context The documentation description of {api_name} is as follows: {api_doc}. The code of {api_name} is: {api_code}. Value Space of each Parameter: {input_space} Target SRP values: {SRP_values}	
Task 1.Determine whether the target SRP values conflict. 2.If they are compatible, complete all missing parameters and construct a valid test case that preserves the target values. 3.If they conflict, identify the conflicting parameters.	
Output Format(Compatible) <pre>test_case = { 'param-1':value-1, ...}</pre>	Output Format(Conflict) <pre>conflict_params = ['param-1', 'param-2', ...]</pre>

Fig. 4: Prompt template and expected output formats for LLM-assisted test-case completion.

message together with the failing input to identify the parameter values responsible for the failure. If the failure is attributed to an invalid SRP-value combination, this combination is removed from the remaining interaction targets and recorded to prevent equivalent invalid configurations from being generated again.

A failure may also arise from the values assigned to the remaining parameters, while the targeted SRP pair itself is valid. In this case, *ShapeTester* retains the target pair and asks the LLM to complete a new test input with compatible values for the other parameters. The regenerated input is executed again, and this process continues until the target pair is either covered by a valid execution or identified as an invalid combination. In this way, execution failures refine subsequent generation without prematurely discarding SRP interactions that may still be feasible.

3.3.2 Shape-Diversity Guided Evolutionary Search

The first stage of the generation terminates once the set of uncovered SRP pairs is exhausted, ensuring a shallow complete exploration of the parameter interaction space. To explore beyond these shallow interactions and uncover subtle defects arising from complex, multi-parameter dependencies, *ShapeTester* transitions to a second stage centered on an evolutionary search strategy.

The output tensor shape provides an observable structural signature of an API execution. Inputs that induce different output ranks or dimension profiles may exercise different branches of the framework’s shape-calculation logic. We therefore use output-shape variation to guide the second-stage search. Comparing output shapes requires accounting for the fact that the same API may produce tensors of different ranks under different parameter configurations. Conventional vector distances are not directly applicable to such variable-length shape tuples without introducing a padding or axis-alignment rule, which would impose additional assumptions about the correspondence between tensor axes. We instead use a rank-aware similarity function: shapes of different ranks are assigned zero similarity, while shapes of the same rank are compared according to their relative dimension profiles using cosine similarity:

$$Sim(S_1, S_2) = \begin{cases} 0.0, & \text{if } \text{Dim}(S_1) \neq \text{Dim}(S_2) \\ \frac{\mathbf{S}_1 \cdot \mathbf{S}_2}{\|\mathbf{S}_1\| \|\mathbf{S}_2\|}, & \text{otherwise} \end{cases} \quad (1)$$

Here, $\mathbf{S}_i$ is the ordered vector of dimension sizes in shape S_i , and $\text{Dim}(S_i)$ denotes its rank. This definition distinguishes both rank changes and reordered dimension profiles. For example, $Sim((1, 1000), (1000, 1))$ is approximately 0.002, whereas shapes of different ranks, such as $(1000, 1)$ and $(1000, 1, 1)$, receive zero similarity.

We use this similarity to discourage the evolutionary search from repeatedly selecting inputs that produce closely aligned output-shape profiles. Specifically, the fitness of a candidate input is defined as inversely proportional to its similarity to existing population members:

$$fitness(S_i) = \frac{1}{\sum_{j \neq i} e^{Sim(S_i, S_j)}} \quad (2)$$

This formulation encourages the search to prioritize inputs that produce under-explored output-shape profiles rather than repeatedly sampling structurally similar outputs.

The search starts from a population initialized with the valid test inputs generated in Stage 1. Each individual represents a complete API test input as a mapping from parameter names to concrete values and is associated with the output shape obtained from its successful execution. At the beginning of each generation, *ShapeTester* recomputes the diversity-oriented fitness scores of the population and selects the two individuals producing the most distinct output-shape profiles as parents.

ShapeTester then applies uniform parameter-level crossover: for each parameter position shared by the two parents, their values are exchanged independently with crossover probability p_c . The resulting offspring are subsequently mutated by independently replacing each parameter value, with mutation probability p_m , with a value sampled from its current candidate domain. This process produces at most four offspring per generation.

Before execution, duplicate offspring and offspring matching an accumulated invalid parameter combination are discarded and regenerated. Candidates rejected by this constraint filtering do not consume an execution from the testing budget. The remaining candidates are evaluated using the cross-mode oracle. Successfully executed candidates are added to the population together with their output shapes, cross-mode inconsistencies are recorded as potential defects, and invalid executions are excluded from the population. When the population exceeds its configured maximum size, the oldest individuals are removed. The updated population then provides both the candidate parents and shape information for the next generation. This feedback-guided evolutionary loop progressively bypasses shallow validity boundaries and focuses the search on deeper semantic regions where complex shape inconsistencies are more likely to manifest.

3.4 Execution and Shape Consistency Oracle

The final component of *ShapeTester* is responsible for validating generated test inputs and detecting shape errors through a differential oracle.

3.4.1 Cross-Mode Differential Oracle

Our design is based on the observation that modern deep learning frameworks typically provide both dynamic and static execution paths that are intended to implement the same API-level tensor transformation. For a given API and test configuration, the two paths should therefore produce consistent output shapes.

Given a generated test case, *ShapeTester* constructs dynamic and static executions from the same API wrapper and parameter configuration. The values assigned to the API parameters remain unchanged between the two executions. Only the representation of the input tensors and the framework-specific execution interface differ. The dynamic path produces a concrete runtime output shape, denoted by S_{dyn} , whereas the static path produces a graph or symbolic output shape, denoted by S_{stat} .

TensorFlow. For TensorFlow APIs, *ShapeTester* first invokes the API wrapper directly with the generated concrete tensors and extracts the shape of the returned eager tensor as S_{dyn} . It then wraps the same API function using `tf.function`, invokes the resulting graph function with the same test-case arguments, and extracts the shape of its returned tensor as S_{stat} . Thus, the comparison preserves the tested API, its parameter values, and its concrete inputs while changing the framework execution interface.

Keras. For Keras APIs, *ShapeTester* obtains S_{dyn} by invoking the API wrapper with the generated concrete input tensors. To obtain S_{stat} , it replaces each concrete input tensor with a `Keras Input` that preserves the same non-batch dimensions, while retaining the same API parameters, and invokes the same wrapper symbolically. Because a `Keras Input` represents its batch dimension symbolically, the concrete batch dimension in the dynamic output is normalized to `None` before the two shapes are compared. For example, a dynamic shape of (2, 3, 5) is normalized to (None, 3, 5) before being compared with the corresponding symbolic shape.

PyTorch. For PyTorch APIs, *ShapeTester* obtains S_{dyn} by directly invoking the API wrapper in eager mode. It then compiles the same wrapper using `torch.compile(..., dynamic=True)`, invokes the compiled function with the same test-case arguments, and extracts the shape of the returned tensor as S_{stat} . The comparison therefore checks whether direct eager execution and the compiled execution path produce the same tensor shape under an identical API configuration.

Shape extraction and comparison. Across the three frameworks, *ShapeTester* recursively extracts the complete shape tuple from each returned tensor object. For APIs with multiple outputs, all corresponding output shapes are retained and compared in their return order. After the framework-specific batch-dimension normalization described above, shape equality is determined structurally. Specifically, the two results must contain the same number and nesting structure of outputs, and every pair of corresponding outputs must have the same rank and identical dimension values at each position. An unknown dimension is not treated as a wildcard for a different concrete dimension after the applicable normalization.

Each execution path records either an output shape or an exception. Shape equality is evaluated only when both paths successfully return output shapes. Based on the two recorded results, *ShapeTester* classifies the test outcome into the following three categories:

- **Consistent Pass:** Both modes execute successfully and their normalized output-shape structures are identical ($S_{dyn} == S_{stat}$), indicating consistent shape-transformation behavior.
- **Equivalent Failure:** Both modes raise runtime exceptions, suggesting that the generated input violates validity constraints shared by the two execution paths. The failure information is subsequently provided to the failure-driven constraint-inference component.
- **Shape Inconsistency:** A potential defect is reported when cross-mode consistency is violated. This category includes two situations: (1) both modes execute successfully but produce different normalized output shapes ($S_{dyn} \neq S_{stat}$), or (2) one mode successfully returns an output shape while the other mode raises an exception.

Through this procedure, the oracle compares the complete shape structures produced by two execution paths for the same API configuration. It thereby captures shape inconsistencies that may not cause an immediate crash in ordinary dynamic execution. Since the dynamic and static execution paths are expected to implement equivalent shape-transformation semantics, any inconsistency between their results violates this fundamental cross-mode consistency and is therefore reported as a potential defect. Importantly, the oracle does not assume that either execution path is inherently correct. A mismatch indicates that at least one path deviates from the expected API semantics: it may result from an incorrect static shape calculation, an error in dynamic execution, or a limitation in the implementation of either execution mode. These cases all fall within the class of cross-mode shape defects targeted by *ShapeTester*.

3.4.2 Failure-Driven Constraint Inference

Beyond error detection, *ShapeTester* leverages execution failures to refine the input generation process. Instead of treating failures as mere invalid cases, we interpret them as evidence of violated input constraints.

To this end, *ShapeTester* employs a diagnostic component to infer fine-grained parameter constraints from failure traces. Given an execution error, *ShapeTester* leverages an LLM to analyze the error message and identify the parameter interactions responsible for the violation. This process yields invalid parameter combinations that approximate the implicit input contract enforced by the framework.

As shown in Figure 5, the failure-analysis prompt contains the framework name, API documentation, wrapper code, failing test case, and runtime error message. The expected response follows a prescribed Python format containing a `param_comb` list of the parameters jointly responsible for the failure. Before incorporating the result into the constraint set, *ShapeTester* verifies that the response is parsable and that every returned name occurs in the wrapper’s parameter list.

Once the responsible parameters have been identified, *ShapeTester* records their values in the failing input as an invalid combination and feeds this information back to test generation. During SRP-guided generation, the recorded combinations are used to filter candidate completions and to determine whether a failed target pair

Role
You are a deep learning engineer proficient in the {frame_name} framework.
Context
The documentation description of {api_name} is as follows: {api_doc}.
The code of {api_name} is: {api_code}.
Failing test case: {test_case}
Runtime error: {error_message}
Task
Analyze the failure and identify the combination of parameters responsible for the error.
Output Format
param_comb = ['param-1', 'param-2', ...]

Fig. 5: Prompt template and expected output format for failure-driven constraint extraction.

should be excluded or retried with different values for the remaining parameters. During shape-diversity guided evolutionary search, each offspring produced by crossover and mutation is checked against the accumulated invalid combinations. An offspring containing a known invalid combination is discarded and regenerated before execution.

The remaining candidates are evaluated by the cross-mode oracle. Valid executions contribute their output shapes to the evolving population, while failed executions provide additional feedback for updating the constraint set. This iterative process connects concrete execution outcomes with both combinatorial exploration and evolutionary search, allowing *ShapeTester* to progressively avoid repeatedly encountered invalid regions and devote more tests to executable shape transformations.

4 Evaluation

To evaluate the effectiveness of *ShapeTester*, we investigate the following research questions (RQs):

- **RQ1:** How many shape errors can *ShapeTester* detect in real-world popular deep learning frameworks?
- **RQ2:** How does *ShapeTester* compare to state-of-the-art baseline approaches in terms of shape error detection rate and output shape diversity?
- **RQ3:** What is the individual contribution of each core component to the overall effectiveness of *ShapeTester*?
- **RQ4:** What are the computational costs of *ShapeTester* compared with existing approaches, and how are its time and token costs distributed across different stages?

The machine for running experiments is equipped with an Intel Xeon Gold 6330 CPU (28 cores, 2.00GHz), Ubuntu 24.04, and Python 3.10.

4.1 Evaluation Setup

Before the evaluation, we introduce the implementation settings of our approach and the construction of our benchmark dataset, as well as the baselines and metrics used in the evaluation.

4.1.1 Implementation

We implemented the prototype of *ShapeTester* using approximately 2,500 lines of Python code. Following the evaluation practice of prior LLM-based DL testing studies, which instantiate their LLM-assisted components under a fixed model configuration [8, 9], we used the same model for all LLM-assisted tasks. Since these tasks require reasoning over API documentation, wrapper code, parameter dependencies, and run-time diagnostics, we selected the reasoning-oriented DeepSeek-R1 model [12]. We set the temperature to 0 and left the remaining generation parameters at their server-side defaults. In particular, the maximum output length was 4,096 tokens [13]. The client-side timeout for each request was 300 seconds, and each prompt was attempted at most three times, including the initial request.

Each response is extracted according to its prescribed format and validated before being used. Generated Python content must contain the required objects and satisfy the corresponding structural checks, whereas JSON responses must be parsable and contain only parameter names defined by the API wrapper. If extraction or validation fails, *ShapeTester* appends the validation error to the prompt and issues another request, with at most three attempts in total. The LLM-derived information is used to improve test generation and prune likely invalid regions. Therefore, exhausting these attempts does not terminate testing; *ShapeTester* continues with the available search-space information and previously generated valid inputs. For the shape-diversity guided evolutionary search stage, we set both the parameter-level crossover probability p_c and mutation probability p_m to 0.2, following established practices in search-based software testing [14]. Each generation produces at most four offspring, and the population is capped at 500 individuals.

4.1.2 Benchmarks

As *ShapeTester* is the first testing methodology specifically for shape-related defects in DL APIs, no off-the-shelf benchmark currently exists for this task. We therefore constructed a benchmark of 100 distinct APIs based on the findings of our initial bug-detection campaign. We first collected APIs associated with developer-confirmed issues and treated each API as a single benchmark subject, regardless of the number of issues reported for it. From the eligible APIs, we randomly sampled 41 TensorFlow APIs and 41 PyTorch APIs to obtain a balanced set of 82 bug-triggering APIs.

To broaden the benchmark beyond APIs with observed failures, we further included nine APIs from each framework for which *ShapeTester* detected no shape discrepancy. These 18 comparison APIs were selected from functional families represented in the bug-triggering set by considering similarities in API names and tensor operations. For example, if `Conv2D` was included as a bug-triggering API, related variants such as `Conv1D` and `Conv3D` were considered as comparison APIs. Although

ShapeTester detected no shape discrepancy for these APIs during its initial campaign, subsequent targeted manual testing, guided by their similarity to the bug-triggering APIs, identified one ground-truth shape bug in each comparison API. Thus, the term comparison API distinguishes their selection and bug-discovery process rather than indicating the absence of bugs. The resulting benchmark therefore combines bugs detected by *ShapeTester* with bugs in closely related APIs initially missed by *ShapeTester*, while maintaining a balanced framework distribution.

The resulting 100-API benchmark is used in full for RQ3, which evaluates *ShapeTester* and its ablated variants. RQ2, however, compares *ShapeTester* with four independently implemented baselines whose applicable API sets differ. We therefore examined the API compatibility of each baseline before conducting the comparative experiment. An API was considered supported when the baseline could use its original test-generation procedure to generate and execute test invocations for that API under the evaluated tool and framework configurations. As summarized in Figure 6, DeepREL, NablaFuzz, TitanFuzz, and MirrorFuzz support 34, 46, 70, and 79 of the 100 APIs, respectively. Their common intersection contains 20 APIs, including 14 bug-triggering APIs and 6 comparison APIs; all 20 are associated with a known shape bug, with the bugs in the comparison APIs identified through subsequent manual testing. We use this intersection as the RQ2 benchmark so that every approach is evaluated on the same API subjects. The intersection was determined solely from API compatibility before examining the bug-detection, validity, or shape-diversity results. Accordingly, RQ2 characterizes the comparative performance of the evaluated approaches within their common API-support scope, whereas RQ3 examines *ShapeTester*’s components over the complete benchmark.

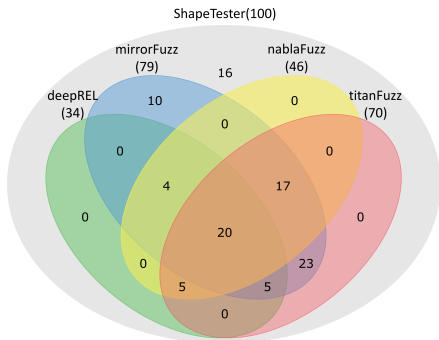


Fig. 6: API-support sets of the four evaluated baselines within the 100-API benchmark. The common intersection contains the 20 APIs used in RQ2.

Bug-triggering APIs (14)	
1. torch.nn.ConvTranspose2d	8. torch.nn.MaxUnpool1d
2. torch.clamp	9. torch.nn.Conv2d
3. torch.empty	10. tf.keras.layers.Conv2D
4. torch.matmul	11. tf.keras.layers.AveragePooling2D
5. torch.sparse_coo_tensor	12. tf.keras.layers.Conv3D
6. torch.nn.InstanceNorm1d	13. tf.keras.layers.DepthwiseConv2D
7. torch.nn.InstanceNorm3d	14. tf.keras.layers.SeparableConv2D
Comparison APIs (6)	
1. torch.as_strided	4. torch.nn.MaxUnpool2d
2. torch.gather	5. tf.keras.layers.BatchNormalization
3. torch.nn.Softmin	6. tf.keras.layers.LSTM

Fig. 7: The 20 common APIs in the intersection.

4.1.3 Baselines and Ablation Methods

To the best of our knowledge, *ShapeTester* is the first shape-error-targeted testing approach for DL APIs. Existing DL framework testing approaches do not provide a test oracle specifically designed for shape inconsistencies. To evaluate their test-generation capabilities in the shape-error-testing setting, we therefore equipped each baseline with *ShapeTester*’s shape oracle. Because this oracle evaluates the dynamic and static or symbolic output shapes of individual API invocations, we considered API-level testing approaches for the comparison. To provide representative coverage of recent API-level DL framework testing techniques, we selected both traditional and LLM-based state-of-the-art approaches. Specifically, DeepREL and NablaFuzz represent traditional API-level techniques based on relational API testing and automatic-differentiation testing, respectively, while TitanFuzz and MirrorFuzz represent LLM-based test-generation techniques [6, 7, 8, 9]. This selection covers different test-generation paradigms under a common API-level evaluation setting.

During the evaluation, each baseline retained its original test-generation procedure and internal guidance mechanisms. The generated API invocations were then executed and evaluated using *ShapeTester*’s shape oracle, which compares the output shapes obtained from the corresponding dynamic and static or symbolic execution paths. Applying the same oracle to all generated inputs provides a consistent basis for assessing how effectively the different generation strategies exercise shape-related behaviors.

DeepREL[6] infers relational APIs within deep learning libraries by analyzing API signatures and documentation. It leverages these relations as test oracles, enabling differential testing to detect inconsistencies between these relational APIs.

NablaFuzz[7] targets the automatic differentiation (AD) components of DL libraries. It performs differential testing across various AD modes (e.g., forward and reverse modes) and numerical differentiation to detect inconsistencies in both first- and higher-order gradient computations.

TitanFuzz[8] directly leverages large language models (LLMs) to generate and mutate input programs for DL libraries. It combines a generative LLM for seed creation with an infilling LLM for mutation to maximize code coverage and API diversity.

MirrorFuzz[9] exploits the observation that similar APIs within or across DL frameworks often share similar bugs. It collects historical bug data, matches operationally or parametrically similar APIs, and prompts LLMs to synthesize test cases that reveal analogous bugs in the API under test.

For RQ3, we conduct a comprehensive evaluation to assess the impact of *ShapeTester*’s three core strategies: SRP Guided Generation, Shape-Diversity Guided Evolutionary Search, and Constraints-aware feedback. To quantify their contributions, we designed three ablation variants.

ShapeTester^{w/o.Comb} removes the SRP guided test input generation loop and leverages only the evolutionary search stage to generate test inputs.

ShapeTester^{w/o.GA} removes the evolutionary search stage and uses only the SRP-guided generation stage to generate test inputs.

ShapeTester^{w/o.Constr} removes the guidance of constraints.

4.1.4 Evaluation Metrics

To rigorously assess the performance of *ShapeTester*, we employ three primary metrics to evaluate its error-detection capability, its efficiency in navigating API constraints, and its ability to explore the tensor transformation space.

Number of Detected Bugs. The primary objective of *ShapeTester* is the identification of shape errors. For *RQ1*, every candidate issue was first reproduced under the evaluated framework version. A researcher then checked whether the input satisfied the documented constraints and whether the observed dynamic–static shape discrepancy violated the API’s documented or intended shape semantics. This assessment was independently cross-checked by another researcher before the issue was submitted to the framework developers. We report a candidate as a human-verified issue only after this two-stage inspection; the *Confirmed* and *Fixed* categories in Table 1 additionally reflect the subsequent status assigned by framework developers. For *RQ2* and *RQ3*, by contrast, the number of detected bugs is an automated experimental metric: each unique dynamic–static shape discrepancy is counted once under a given configuration. These comparative counts are therefore used to compare testing configurations. For *RQ2*, each of the 20 API subjects is associated with one ground-truth shape bug, and the *Bug Detection Rate* is defined as the number of these bugs detected by an approach divided by 20. Crucially, we have integrated our specialized shape oracle into all baseline methodologies. By granting baselines the same detection capabilities as *ShapeTester*, we can isolate and evaluate the intrinsic effectiveness of each approach’s test generation strategy in exercising complex shape-transformation logic.

Validity Rate. A key challenge in DL API testing (Challenge 1) is the presence of complex parameter constraints that trigger early execution crashes. We define the *Validity Rate* as the ratio of test inputs that successfully execute without triggering any crash to the total number of generated inputs. A higher validity rate demonstrates the effectiveness of the constraint-aware feedback in bypassing validation checks to reach the core shape-calculation logic.

Shape Diversity. Beyond execution validity, we evaluate how broadly each approach explores the output-shape space. We adopt the similarity-based spectral formulation underlying the Vendi Score [15], a general diversity framework that summarizes a collection through the eigenspectrum of a user-defined similarity matrix. Let $\{s_1, s_2, \dots, s_n\}$ be the set of n unique output shapes discovered during testing. Using the rank-aware cosine similarity defined in Equation 1, we construct a normalized similarity matrix $\rho \in \mathbb{R}^{n \times n}$:

$$\rho_{i,j} = \frac{\text{sim}(s_i, s_j)}{n} \quad (3)$$

The Von Neumann Entropy (VNE) is then calculated using the eigenvalues λ_i of ρ :

$$S(\rho) = - \sum_{i=1}^n \lambda_i \ln \lambda_i \quad (4)$$

where λ_i represents the i -th eigenvalue of ρ . A higher $S(\rho)$ indicates that the observed shapes are distributed across a broader set of mutually dissimilar rank and dimension

profiles under the adopted similarity function. We refer to this quantity as output-shape diversity in the remainder of the evaluation.

Computational Efficiency. We evaluate computational efficiency using the average time cost per API and, for LLM-based approaches, the average token cost per API. All approaches are evaluated on the same machine and the same 20 APIs used in RQ2, with 100 executed test inputs generated for each API. The time cost covers the complete testing process, including initialization, test-input generation, LLM invocations when applicable, and the execution of generated tests. The token cost includes both input and output tokens. Because DeepREL and NablaFuzz do not invoke an LLM, token cost is not applicable to these two approaches. For *ShapeTester*, we further analyze its token cost according to the purposes of LLM calls and its time cost according to the main testing stages.

4.2 RQ1. Bugs Detected

To evaluate the practical effectiveness of *ShapeTester*, we deployed it against popular DL frameworks Keras (v3.3.3 and v3.10.0), TensorFlow (v2.16.0 and v2.21.0), and PyTorch (v2.4.0 and v2.10.0). Table 1 summarizes real-world bugs detected by *ShapeTester* across three popular DL frameworks and categorizes the detected issues according to their confirmation status and error manifestation. The *Confirmed* column reports the number of issues that developers have acknowledged as previously unknown, while the *Fixed* column indicates those that have been subsequently resolved. The *Pending* column captures issues awaiting developer review. In addition, we classify each issue by its observable behavior: *Crash* refers to unexpected runtime failures that occur despite the input satisfying parameter constraints, whereas *Shape Error* denotes silent failures where the API executes without crashing but produces an output tensor whose dimensions deviate from the expected shape semantics.

Table 1: Bugs detected by *ShapeTester* in TensorFlow, Keras, and PyTorch.

	Total	Status			Error Type	
		Pending	Confirmed	Fixed	Crash	Shape Error
TensorFlow	16	0	13	3	12	4
Keras	69	5	12	52	18	51
PyTorch	35	0	18	17	1	34
Total	120	5	43	72	31	89

In total, we identified and reported 120 previously unknown bugs to the respective framework maintainers. As of the time of writing, 95.8%(115/120) of the reported bugs have been officially confirmed or fixed. Unlike traditional fuzzing approaches that rely on crash-based oracles, which often suffer from high false-positive rates due to invalid input permutations, *ShapeTester* leverages its differential shape oracle to pinpoint precise logical inconsistencies. This specialized oracle helps reduce manual effort in bug triage, as discrepancies between dynamic execution and static inference

typically indicate fundamental flaws in the framework’s internal logic rather than mere environmental noise.

Notably, 89 out of 120 detected bugs are shape errors. This high proportion aligns with the core objective of *ShapeTester*, which is specifically designed to target shape-related defects. Crucially, these shape errors would largely remain undetected by existing testing approaches, as such methods lack a dedicated oracle capable of identifying errors in shape computation.

The distribution of detected errors varies substantially across the three frameworks, reflecting differences in how static and dynamic shapes are computed. Keras serves as a multi-backend frontend for TensorFlow, PyTorch, and JAX [16]. For the Keras layer APIs examined in our study, symbolic output shapes are computed by Keras, commonly through `compute_output_shape()`, whereas concrete output shapes are produced by the corresponding backend. Consequently, when these APIs use the TensorFlow backend, errors in static shape computation are generally attributed to Keras, while errors in dynamic shape computation are attributed to TensorFlow. For example, the static shape error of `Conv2D` was reported to Keras (issue #22495²), whereas its dynamic shape error was reported to TensorFlow (issue #64396³). This separation helps explain why more shape errors in Table 1 are associated with Keras than with TensorFlow. PyTorch presents a contrasting pattern, with 35 total errors, of which 34 are shape errors and only one is a crash. This concentration is largely attributable to its meta-device and Dynamo mechanisms, which perform symbolic shape inference during graph compilation independently of the dynamic execution path.

Answer to RQ1. In real-world DL frameworks, *ShapeTester* successfully detects 120 previously unidentified bugs, of which 115 (95.8%) have been confirmed or fixed by framework developers. Among these detected issues, 89 (74.1%) are silent shape errors, which depend on the proposed differential oracle for detection.

4.3 RQ2: Comparison with State-of-the-Art Baselines

We compare *ShapeTester* with *DeepREL*, *NablaFuzz*, *TitanFuzz*, and *MirrorFuzz* on the common-support 20-API benchmark described in the evaluation setup. Because these tools were originally designed for different testing objectives and employ different native oracles, we evaluate their generated tests using the same shape-error oracle implemented in *ShapeTester*. This controlled setup isolates their test-generation strategies and allows the reported metrics to characterize how effectively each approach exercises shape-related behaviors. Since APIs differ in the number of SRPs, domain sizes, and parameter constraints, the number of test inputs required to exercise their input spaces also varies. A small number of inputs may cover the interaction space of an API with a compact parameter space, whereas APIs with larger parameter spaces generally require more inputs. To provide every evaluated approach with the same testing budget, consistent with the fixed per-API evaluation practice in prior API-level DL library testing [6], we configure each approach to generate 100 test inputs per API and evaluate them using three metrics: bug detection rate, valid input rate, and shape diversity. For every approach, test generation for an API terminates immediately after

²<https://github.com/keras-team/keras/issues/22495>

³<https://github.com/tensorflow/tensorflow/issues/64396>

100 test inputs have been executed; candidates rejected by constraint filtering before execution are not counted toward this budget. Thus, the 20 APIs refer to the API subjects included in the comparative benchmark rather than the total number of test inputs. Because these APIs constitute the complete common-support intersection of the four baselines, each approach is evaluated on all eligible API subjects and executes a total of 2,000 test inputs across them.

Table 2: Comparison of *ShapeTester* and state-of-the-art baselines on the 20-API evaluation subset.

	DeepREL	NablaFuzz	TitanFuzz	MirrorFuzz	ShapeTester
Bug Detection Rate	0.45	0.50	0.45	0.50	0.70
Valid Input Rate	0.42	0.46	0.74	0.39	0.80
Shape Diversity	0.65	0.73	0.65	0.62	0.74

Effectiveness in Shape Error Detection. As summarized in Table 2, *ShapeTester* achieved the highest observed values across all three metrics. In terms of bug detection, *ShapeTester* achieves a detection rate of 0.70 (14/20 ground-truth shape bugs), which represents a 40% improvement over the strongest baselines, *NablaFuzz* and *MirrorFuzz* (both at 0.50). This performance gap is further illustrated in Figure 8, which depicts the cumulative number of unique shape errors discovered as the number of test cases increases. *ShapeTester* (the purple curve) exhibits a steeper discovery slope and a higher saturation point compared to all baselines. While *MirrorFuzz* and *NablaFuzz* reach a plateau early in the testing process, *ShapeTester* continues to uncover deep-seated bugs, validating that the combination of SRP-interaction guided generation and evolutionary search is more effective at reaching complex shape-calculation logic than generic mutation or historical pattern matching.

Analysis of Execution Validity. The efficacy of a shape-testing tool is heavily contingent on its ability to bypass shallow parameter validation checks. Our results show that *ShapeTester* achieves the highest validity rate of 0.80, outperforming *TitanFuzz* (0.74) and nearly doubling the validity of *MirrorFuzz* (0.39) and *DeepREL* (0.42). The low validity rates of traditional fuzzers suggest that they frequently generate illegal parameter combinations that trigger immediate **ValueErrors** or **TypeErrors**. In contrast, the constraint-aware feedback loop in *ShapeTester* successfully utilizes runtime error messages to dynamically prune the invalid search space, thereby ensuring that a higher proportion of computational resources is spent on executing valid API calls that exercise the actual tensor transformation kernels.

Exploration of Shape Diversity. Regarding the structural diversity of generated output shapes, *ShapeTester* achieves the highest value of 0.74 among all evaluated tools. Notably, *NablaFuzz* also maintains high diversity (0.73) due to its focus on automatic differentiation modes. However, it fails to translate this diversity into a higher bug detection rate (0.50). These results suggest that output-shape diversity alone is not sufficient for detecting shape errors: interactions among shape-related

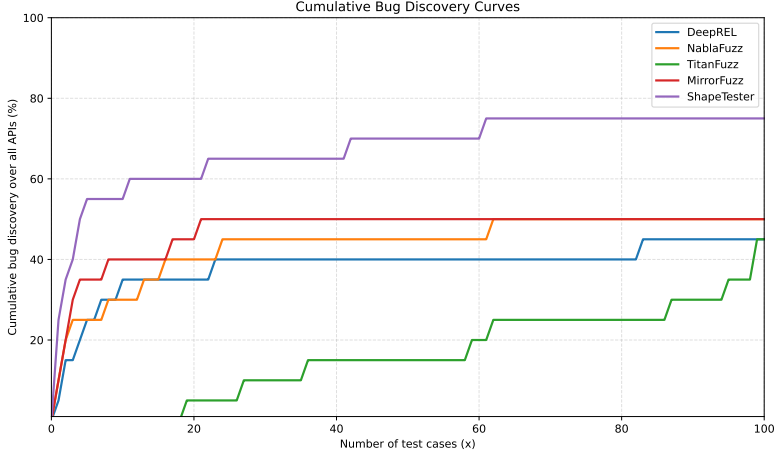


Fig. 8: Cumulative bug discovery curves over all evaluated APIs for *ShapeTester* and baselines.

parameters also determine whether generated inputs reach fault-triggering configurations. By combining interaction-oriented generation with diversity-guided evolutionary search, *ShapeTester* explores varied output-shape profiles while systematically exercising SRP interactions.

Answer to RQ2. *ShapeTester* achieves a 0.70 bug detection rate, which is 40% higher than the 0.50 achieved by the strongest existing methods. This effectiveness is accompanied by the highest execution validity rate (0.80) and output-shape diversity (0.74), indicating broad exploration of structurally different output shapes under the adopted metric. The cumulative bug discovery curves further show that *ShapeTester* reaches the same cumulative bug counts with fewer executed test cases and attains a higher saturation point than the evaluated SOTA testing techniques.

4.4 RQ3: Ablation Study

To investigate the contribution of each core component of *ShapeTester*, we evaluate the full approach and three ablation variants $ShapeTester^{w/o-GA}$, $ShapeTester^{w/o-Comb}$, and $ShapeTester^{w/o-Constr}$ on the same 100-API benchmark. The results are summarized in Table 3.

Table 3: Ablation study results of *ShapeTester* (ST) and its three variants.

	Full ST	$ST^{w/o-Comb}$	$ST^{w/o-GA}$	$ST^{w/o-Constr}$
Bugs Detected	82	73	71	77
Validity Rate	0.761	0.752	0.771	0.706
Shape Diversity	0.421	0.269	0.340	0.333

Contribution of SRP Guided Generation. The comparison between the full *ShapeTester* and *ShapeTester^{w/o-Comb}* reveals that the combinatorial testing stage is fundamental to the exploration of the shape transformation space. Without the initial SRP-interaction guided seeds, the shape diversity drops from 0.421 to 0.269, the lowest among all variants. This reduction in diversity directly leads to a decrease in bug detection, with 9 fewer bugs identified. The results suggest that systematic coverage of shape-related parameter interactions provides a critical foundation that random or purely evolutionary methods cannot easily replicate, especially when navigating high-dimensional API parameter spaces.

Contribution of Shape-Diversity Guided Evolutionary Search. The removal of the evolutionary search stage (*ShapeTester^{w/o-GA}*) results in the most substantial decline in bug detection capability, falling from 82 to 71 detected bugs. While *ShapeTester^{w/o-GA}* maintains a slightly higher validity rate (0.771), its shape diversity (0.340) is notably lower than the full version. This indicates that while combinatorial testing is efficient at generating valid inputs, it may fail to reach unprecedented corner cases that reside deep within the shape-calculation logic. The output-shape-diversity objective encourages the evolutionary stage to retain inputs that expose distinct rank and dimension profiles. Together with parameter-level crossover and mutation, this stage detects 11 additional shape errors compared with the variant without evolutionary search.

Contribution of Constraints-aware Feedback. The constraint-aware feedback loop improves the effectiveness of test generation by preventing repeatedly observed invalid parameter combinations from re-entering the search. When this component is removed (*ShapeTester^{w/o-Constr}*), the validity rate decreases from 0.761 to 0.706, and the number of detected bugs decreases from 82 to 77. Without failure-driven refinement, more generated inputs terminate at parameter validation checks before exercising the underlying shape-transformation logic. These results demonstrate that feeding diagnosed failures back to both generation stages improves execution validity and preserves more of the testing budget for shape-related behaviors.

Answer to RQ3. Each component of *ShapeTester* plays a distinct and vital role: SRP-interaction coverage provides the breadth of initial exploration, shape-diversity guidance enables the discovery of deep-seated bugs through evolutionary search, and the constraint-aware feedback loop ensures high execution validity. The full *ShapeTester* framework leverages the synergy of these strategies to achieve the highest bug detection rate and shape diversity.

4.5 RQ4: Efficiency Analysis

To understand the computational efficiency of *ShapeTester*, we first compare its end-to-end time and token costs with those of the four baseline approaches. We then analyze the distribution of its token cost across different LLM operations and its time cost across the main testing stages.

Overall Efficiency Comparison. Table 4 presents the average computational costs of the evaluated approaches. The measurements were collected using the same experimental machine, the same 20 APIs, and the same number of executed test inputs

per API. Time cost covers the complete testing process. For the three LLM-based approaches, token cost includes both input and output tokens.

Table 4: Computational costs of the evaluated approaches.

	DeepREL	NablaFuzz	TitanFuzz	MirrorFuzz	ShapeTester
Time/API (s)	18.23	7.08	25,990.76	16,069.15	5,968.00
Tokens/API	–	–	133,279.94	1,055,101.80	77,268.82

DeepREL and NablaFuzz have lower time costs because their test-generation procedures do not involve LLM calls. Among the three LLM-based approaches, *ShapeTester* has the lowest time and token costs. Its average time cost is 5,968.00 seconds per API, compared with 16,069.15 seconds for MirrorFuzz and 25,990.76 seconds for TitanFuzz. Its average token cost is 77,268.82 tokens per API, which is lower than the 133,279.94 tokens used by TitanFuzz and the 1,055,101.80 tokens used by MirrorFuzz.

This difference is associated with how the LLM participates in test generation. The main test inputs of *ShapeTester* are constructed through combinatorial testing and evolutionary search. The LLM is invoked only when semantic information is required for initialization, when an incomplete test input must be completed, or when an encountered conflict must be analyzed. By contrast, TitanFuzz and MirrorFuzz invoke LLMs directly during their test-input generation processes. The selective use of the LLM therefore enables *ShapeTester* to maintain lower time and token costs than the evaluated LLM-based approaches.

Token-Cost Analysis Across LLM Operations. We further analyze the token cost of the three types of LLM operations in *ShapeTester*: initialization, test-input completion, and conflict analysis. Table 5 reports the average number of calls, average tokens consumed by each call, total token cost, and proportion of the overall token cost for each operation.

Table 5: Token costs of the LLM operations in *ShapeTester*.

Operation	Calls/API	Tokens/Call	Tokens/API	Proportion
Initialization	1.00	7,095.96	7,095.96	9%
Test-input completion	3.90	3,836.56	14,962.57	19%
Conflict analysis	20.00	2,760.51	55,210.29	71%

Conflict analysis is the primary source of token cost in *ShapeTester*, accounting for approximately 71% of the total. Although a single conflict-analysis call consumes fewer tokens than an initialization call, conflict analysis is invoked much more frequently. During testing, each newly encountered conflict is analyzed so that the corresponding parameter relationships can be extracted and used in subsequent test generation. As a

result, conflict analysis is invoked 20 times per API on average and consumes 55,210.29 tokens.

Initialization is performed once for each API and accounts for approximately 9% of the total token cost. Test-input completion is invoked when the non-LLM generation procedure cannot construct a complete executable input using the currently available information. It is invoked 3.9 times per API on average and accounts for approximately 19% of the total token cost. These results show that the majority of token consumption is associated with repeatedly analyzing conflicts encountered during testing.

Time-Cost Analysis Across Testing Stages. We divide the execution of *ShapeTester* into initialization, combinatorial testing, and evolutionary testing. Table 6 presents the average time cost and its proportion for each stage.

Table 6: Time costs of the testing stages in *ShapeTester*.

Stage	Time/API (s)	Proportion
Initialization	276.58	5%
Combinatorial testing	5,560.38	93%
Evolutionary testing	131.04	2%
Total	5,968.00	100%

The combinatorial testing stage accounts for approximately 93% of the total time cost of *ShapeTester*. Most parameter conflicts are encountered during this early testing stage because combinations of shape-related parameters are systematically explored for the first time. When a conflict is encountered, *ShapeTester* invokes the LLM to analyze the failure information and extract the corresponding constraints. These conflict-analysis calls are therefore concentrated in the combinatorial testing stage.

LLM invocations dominate the time cost because the reasoning model generally requires hundreds of seconds to complete an invocation, whereas the other operations usually complete within a few seconds. Consequently, the repeated conflict-analysis calls performed during combinatorial testing make this stage the primary source of the overall time cost. After the identified constraints and valid test inputs have been accumulated, the evolutionary testing stage can reuse this information and requires substantially fewer additional LLM operations. It therefore accounts for only approximately 2% of the total time cost.

Answer to RQ4. Among the evaluated LLM-based testing approaches, *ShapeTester* has the lowest computational cost, requiring an average of 5,968.00 seconds and 77,268.82 tokens per API. Its selective LLM invocation strategy reduces both time and token costs because its main test inputs are constructed using combinatorial testing and evolutionary search. Within *ShapeTester*, conflict analysis accounts for approximately 71% of the token cost, and the combinatorial testing stage accounts for approximately 93% of the time cost because most conflicts and the corresponding LLM analyses occur during this stage.

5 Related Work

In this section, we review related work on DL library testing and shape-related bug detection.

5.1 DL Library Testing

Extensive research has investigated the testing of DL libraries [2, 17, 4]. These studies can be broadly divided into model-level and API-level approaches according to the granularity of their test inputs.

Model-level testing. One major line of research exercises DL libraries by constructing or mutating complete DL models. CRADLE [17] directly reuses existing DL models and detects implementation inconsistencies in Keras across different library backends. Building on CRADLE, LEMON [2] and Muffin [18] apply model mutation rules to generate more diverse DL models, thereby invoking a broader range of library code and exposing potential library defects. COMET [19] further proposes three coverage criteria to guide model generation. DeepConstr [20] improves the completeness of inferred constraints through complementary-set-guided refinement, while NeuRI [21] collects valid and invalid API traces and applies inductive program synthesis to infer constraints for constructing valid models. Generated models have also been used to test DL compilers [22, 23]. These approaches exercise a broad range of library behaviors through complete model executions. In comparison, *ShapeTester* operates at the individual-API level and focuses specifically on the shape-transformation semantics implemented by DL framework APIs.

API-level testing. API-level approaches test one DL API at a time and therefore provide finer-grained control over API arguments. Early studies mainly focused on constructing executable API calls. DocTer [4] extracts argument constraints from API documentation and randomly generates both valid and invalid inputs based on the extracted constraints. FreeFuzz [5] instruments library functions and mines input constraints from open-source code, including documentation examples, developer-written unit tests, and real-world DL models; it then generates new inputs according to the collected constraints. Acetest [24] analyzes input-validation code in the C/C++ implementation of each API to extract constraints and generate more valid test cases. These approaches focus on deriving input constraints and constructing executable API calls. *ShapeTester* shares the API-level testing granularity but further organizes its search around shape-related parameters and their interactions, with the specific goal of exercising shape-transformation behavior.

Subsequent studies developed specialized test-generation strategies and test oracles for different classes of DL library defects. IvySyn [25] identifies memory-error-triggering constraints in C/C++ kernels and constructs Python-level API inputs that reproduce the corresponding vulnerabilities. Meta [26] defines 18 metamorphic relations for testing widely used DL operators. DeepREL [6] infers relational APIs from API signatures and documentation and detects inconsistencies by comparing different but semantically related APIs. NablaFuzz [7] targets the automatic differentiation (AD) components of DL libraries and performs differential testing across forward-mode AD, reverse-mode AD, and numerical differentiation to expose inconsistencies

in first- and higher-order gradient computations. These approaches develop specialized generation strategies or test oracles for memory safety, metamorphic properties, relational API consistency, and gradient correctness. However, to the best of our knowledge, *ShapeTester* is the first testing approach to use consistency between concrete output shapes obtained through dynamic execution and shapes obtained through static or symbolic execution as a test oracle for detecting shape-related defects in DL framework APIs. Its test generation is correspondingly organized around shape-related parameters and output-shape exploration.

More recently, LLMs have been used in API-level and general-purpose software testing to extract information from documentation and to generate or mutate test inputs [27, 28]. Fuzz4All [29] uses LLMs as a general-purpose input generation and mutation engine to produce diverse and realistic inputs, while FuzzGPT [30] guides LLMs to synthesize unusual programs for effective fuzzing. TitanFuzz [8] uses a generative LLM to create seed programs and an infilling LLM to mutate them, with the goal of increasing code coverage and API diversity. MirrorFuzz [9] collects historical bug data, matches operationally or parametrically similar APIs, and prompts LLMs to synthesize tests intended to reveal analogous defects in the API under test. Collectively, existing DL testing approaches guide input generation using objectives such as input diversity, code coverage, API exploration, relational consistency, gradient discrepancies, or the reuse of historical bug patterns. *ShapeTester* uses the LLM in a different role: it extracts shape-related parameter information to construct an initial search space. Subsequent test generation combines parameter-interaction coverage and failure-derived constraints with an evolutionary search guided by the structural diversity of observed output shapes. To aggregate pairwise shape similarities into a set-level diversity measure, we instantiate the general similarity-based spectral framework underlying the Vendi Score [15].

5.2 Shape-related Bug Detection

Previous empirical studies have identified shape errors as a major class of defects in DL code for model construction and training [31, 32, 33]. Accordingly, several approaches have been proposed to detect or repair shape errors in DL programs [34, 35, 36, 37].

Ariadne [34] applies the WALA static-analysis framework to machine-learning programs that use TensorFlow. Building on Ariadne, Pythia [36] models shape-related library semantics and statically traces tensor shapes to identify violations and error-prone patterns. ShapeTracer [37] traverses program paths, constructs an abstract data-flow computation graph for each path, and invokes a constraint solver to solve the shape constraints introduced by shape operators. PyTea [38] defines intermediate representations for PyTorch programs, statically explores execution paths, and collects the tensor-shape constraints required by each path’s operation sequence. Tensfa [35] and Tensfa2 [39] use LSTM-based models to detect and repair shape bugs in DL programs.

These approaches analyze tensor-shape propagation, constraints, or incompatibilities in DL programs. *ShapeTester* addresses a complementary problem at the framework level: it tests the shape-transformation behavior implemented by individual DL APIs. It generates API inputs and compares concrete runtime shapes with

statically or symbolically inferred shapes. Thus, existing program-level techniques and *ShapeTester* focus on different fault locations—shape-related problems in client programs and shape-calculation defects in framework APIs, respectively.

6 Threats to Validity

In this section, we discuss the potential threats to the validity of our study and the measures we have taken to mitigate them.

Internal Validity. The primary threat to internal validity involves the potential defects in the implementation of *ShapeTester* and the non-determinism inherent in Large Language Models. To ensure the reliability of our tool, we performed extensive unit testing and manual code reviews on the 2,500 lines of Python code, particularly the core shape-calculation and symbolic inference logic. Furthermore, the identification of SRPs relies on an LLM and may therefore be affected by the inherent randomness of LLM outputs, potentially resulting in inaccurate classifications. To mitigate this threat, we employed a temperature setting of 0 during API calls to maximize reproducibility and manually inspected 10% of the identified SRPs. We found that every SRP in the inspected sample was capable of affecting the output shape. This result provides empirical evidence for the accuracy of SRP identification and increases our confidence in the reliability of the extracted SRP spaces. Consistent with the evaluation practice of prior LLM-based DL testing studies, which assess their approaches under a fixed LLM configuration [8, 9], we instantiated all LLM-assisted components of *ShapeTester* with a single model, DeepSeek-R1. Using a fixed model avoids introducing model changes as an additional experimental variable. Nevertheless, we did not evaluate alternative LLMs, and different models may vary in their interpretation of API semantics and their ability to generate valid structured outputs. Another threat concerns the run-to-run variability of the RQ2 comparison. We conducted one complete run for each approach over the 20-API benchmark, with 100 executed test inputs per API. Because the evaluated test-generation procedures contain stochastic components, independent repetitions may produce different metric values. We did not perform repeated trials or inferential statistical analysis; consequently, the current evaluation does not quantify variation across runs. The resource budget used in RQ2 introduces another threat to internal validity. The comparison controls the number of executed test inputs rather than the time cost: every approach executes 100 test inputs for each API. This setting supports the comparison of test-generation results under the same executed-input budget, but approaches with lower test-generation costs may execute more tests under the same time budget. In particular, the efficiency results in RQ4 show that the non-LLM approaches have substantially lower time costs than the LLM-based approaches. The current evaluation reports total time cost but does not measure time to first bug, valid tests per unit time, or bugs detected per hour. Therefore, the reported effectiveness results do not establish the relative performance of the evaluated approaches under an equal-time budget. Future work will complement the fixed-input evaluation with equal-time experiments and time-normalized bug-detection metrics. Another internal threat concerns the hyperparameter settings in the evolutionary stage, such as the mutation probability of 0.2. We fixed these

parameters throughout the evaluation and did not conduct a systematic sensitivity analysis. Consequently, the reported results do not establish robustness across alternative hyperparameter settings.

External Validity. The main threat to external validity relates to the generalizability of our findings across the vast ecosystem of Deep Learning frameworks and APIs. Our benchmark balances TensorFlow and PyTorch APIs and includes developer-confirmed bug-triggering APIs identified by *ShapeTester*. This construction may not fully reflect the distribution of shape-related defects across the complete API populations of the two frameworks. To mitigate this concern, we complemented the bug-triggering set with functionally related comparison APIs for which *ShapeTester* initially detected no discrepancy; subsequent targeted manual testing identified a ground-truth shape bug in each of these APIs. By covering bugs detected by *ShapeTester* and bugs initially missed by *ShapeTester* but identified through manual testing, the benchmark provides a broader basis for evaluating *ShapeTester* and its components. For the comparison with existing approaches in RQ2, the benchmark is further restricted to the 20 APIs in the complete intersection of the four baselines’ support sets. This construction ensures that all approaches are evaluated on the same eligible API subjects, but the resulting comparison may not generalize to APIs that are unsupported by one or more baselines. Furthermore, while we focused on the most recent stable versions of these frameworks, internal logic changes in future updates could affect the performance of our differential oracle. Nevertheless, the fundamental principle of comparing dynamic execution with symbolic inference is algorithm-intrinsic and likely to remain applicable as frameworks evolve toward more complex graph optimization strategies.

Applicability of the Oracle. The applicability of our cross-mode oracle depends on the availability of both dynamic execution and a corresponding static, symbolic, or compiled execution path for the tested API. We acknowledge that such support is not always available or symmetric across APIs and framework versions. If an API exposes only one of the required execution paths, *ShapeTester* cannot apply the cross-mode comparison and therefore does not provide a fallback oracle for that API, because observing a single execution path is insufficient to determine whether its output shape is semantically correct. All APIs included in our evaluation support the required pair of execution paths under the evaluated framework versions. Consequently, this limitation does not affect the reported experimental results.

Construct Validity. Construct validity concerns the degree to which our evaluation metrics, such as Von Neumann Entropy (VNE) and human verification, accurately reflect the effectiveness of shape-error detection. Output-shape diversity is a proxy for the range of shape-calculation behaviors exercised during testing. The adopted similarity function captures differences in tensor rank, dimension order, and relative dimension profiles, but it does not encode the semantic role of each axis or the complete input-to-output transformation. It is also invariant to proportional scaling of all dimensions. We therefore interpret this metric as structural coverage of the observed output-shape space rather than complete semantic coverage of API behavior. To complement this structural metric, we also report the absolute number of

unique, developer-confirmed bugs as a direct measure of utility. Additionally, the manual verification of detected shape inconsistencies introduces the risk of human error or subjectivity. We mitigate this threat through independent cross-checking and by reporting developer confirmation status separately. Nevertheless, pending reports and APIs whose intended shape semantics are not fully documented may remain subject to interpretation.

7 Conclusion

This paper presents *ShapeTester*, a specialized testing framework designed to address the critical yet often overlooked problem of shape errors in DL APIs. By shifting the testing paradigm from generic code coverage to shape-diversity exploration, *ShapeTester* overcomes the limitations of existing API-level fuzzers that rely primarily on crash-based oracles. *ShapeTester* leverages a two-staged, constraint-aware strategy to generate test inputs. The first stage ensures broad interaction coverage through systematic SRP guided generation, while the second stage employs shape-diversity guided evolutionary search to uncover corner cases in deep shape-transformation logic. Furthermore, our differential shape oracle effectively identifies silent inconsistencies by cross-validating tensor dimensions between dynamic and static execution modes.

Extensive evaluation on TensorFlow, PyTorch, and Keras demonstrates that *ShapeTester* achieves higher observed bug detection and output-shape diversity values than the evaluated SOTA approaches. The successful discovery of 120 previously unknown shape-related bugs, 95.8%(115/120) of which have been officially confirmed or fixed, underscores the practical utility of our approach in enhancing the reliability of industrial-grade DL frameworks. As DL systems continue to evolve with increasingly complex API architectures, the automated identification of geometric miscalculations remains essential. Future work will extend the study to broader API populations, framework versions, and additional DL frameworks, and develop complementary oracles for APIs that do not expose comparable dynamic and static execution paths. These extensions would help clarify and broaden the applicability of shape-aware API testing.

8 Data Availability

Our replication package is publicly available at <https://github.com/Shuo-Sun20/ShapeTester>.

References

- [1] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. Toward understanding deep learning framework bugs. *ACM Trans. Softw. Eng. Methodol.*, 32(6):135:1–135:31, 2023.
- [2] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 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*, pages 788–799, 2020.

- [3] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. Eagle: Creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering*, pages 798–810, 2022.
- [4] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–188, 2022.
- [5] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*, pages 995–1007, 2022.
- [6] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 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*, pages 44–56, 2022.
- [7] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. Fuzzing automatic differentiation in deep-learning libraries. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1174–1186. IEEE, 2023.
- [8] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023.
- [9] Shiwen Ou, Yuwei Li, Lu Yu, Chengkun Wei, Tingke Wen, Qiangpu Chen, Yu Chen, Haizhi Tang, and Zulie Pan. Mirrorfuzz: Leveraging llm and shared bugs for deep learning framework apis fuzzing. *IEEE Transactions on Software Engineering*, 2025.
- [10] Chuan Luo, Shuangyu Lyu, Wei Wu, Hongyu Zhang, Dianhui Chu, and Chunming Hu. Towards high-strength combinatorial interaction testing for highly configurable software systems. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 1579–1591. IEEE, 2025.
- [11] Chuan Luo, Shuangyu Lyu, Qiyuan Zhao, Wei Wu, Hongyu Zhang, and Chunming Hu. Beyond pairwise testing: Advancing 3-wise combinatorial interaction testing for highly configurable systems. In Maria Christakis and Michael Pradel, editors, *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 641–653. ACM, 2024.
- [12] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.
- [13] DeepSeek-AI. Deepseek api documentation. DeepSeek API documentation, 2026. Accessed: 2026-08-03.
- [14] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 168–172. ACM/IEEE, 2022.

- [15] Dan Friedman and Adji Bousso Dieng. The vendi score: A diversity evaluation metric for machine learning. *Transactions on Machine Learning Research*, 2023.
- [16] Keras Team. Keras 3 API documetation, 2024.
- [17] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038. IEEE, 2019.
- [18] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1418–1430, 2022.
- [19] Meiziniu Li, Jialun Cao, Yongqiang Tian, Tsz On Li, Ming Wen, and Shing-Chi Cheung. Comet: Coverage-guided model generation for deep learning library testing. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–34, 2023.
- [20] Gwihwan Go, Chijin Zhou, Quan Zhang, Xiazijian Zou, Heyuan Shi, and Yu Jiang. Towards more complete constraints for deep learning library testing via complementary set guided refinement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1338–1350, 2024.
- [21] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. Neuri: Diversifying dnn generation via inductive rule inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 657–669, 2023.
- [22] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 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*, pages 530–543, 2023.
- [23] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–26, 2022.
- [24] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. Acetest: Automated constraint extraction for testing deep learning operators. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 690–702, 2023.
- [25] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P Kemerlis. {IvySyn}: Automated vulnerability discovery in deep learning frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2383–2400, 2023.
- [26] Jinyin Chen, Chengyu Jia, Yunjie Yan, Jie Ge, Haibin Zheng, and Yao Cheng. A miss is as good as a mile: Metamorphic testing for deep learning operators. *Proceedings of the ACM on Software Engineering*, 1(FSE):2005–2027, 2024.
- [27] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024.

- [28] Lenz Belzner, Thomas Gabor, and Martin Wirsing. Large language model assisted software engineering: prospects, challenges, and a case study. In *International Conference on Bridging the Gap between AI and Reality*, pages 355–374. Springer, 2023.
- [29] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
- [30] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.
- [31] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 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*, pages 510–520, 2019.
- [32] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 1110–1121, 2020.
- [33] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–31, 2023.
- [34] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–10, 2018.
- [35] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. Tensfa: detecting and repairing tensor shape faults in deep learning systems. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–21. IEEE, 2021.
- [36] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. Static analysis of shape in tensorflow programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.
- [37] Chen Liu, Jie Lu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. Detecting tensorflow program bugs in real-world industrial environment. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 55–66. IEEE, 2021.
- [38] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. A static analyzer for detecting tensor shape errors in deep neural network training code. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 337–338, 2022.

- [39] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. Automatically repairing tensor shape faults in deep learning programs. *Information and Software Technology*, 151:107027, 2022.