CaMFuzz: Testing Distributed Training Frameworks via Abundant Configuration–Model Interactions

Zhen Yan
Tsinghua University
China

Yuanliang Chen*
Renmin University of China
China

Fuchen Ma
Tsinghua University
China

Ming Yan
Tsinghua University
China

Zehong Yu
Tsinghua University
China

Zhensheng Xian
Tsinghua University
China

Yu Jiang*
Tsinghua University
China

Abstract

Distributed training frameworks coordinate complex mechanisms such as gradient synchronization, pipeline scheduling, and mixed-precision computation to train user-defined model structures (e.g., Transformers, MoE, LoRA) across multiple GPUs and distributed workers. Even under single-host, multi-GPU execution, interactions between training configurations and model structures can activate diverse execution paths, making distributed training frameworks particularly error-prone. However, existing testing techniques rarely explore configuration–model interactions and therefore miss a large space of Distributed Training Bugs (DTBs).

This paper first studies 133 real-world DTBs across four widely used frameworks, finding that 93% require valid configurations with explicit configuration–model dependencies and 45% further need specific model structures that activate faulty paths via implicit interactions. Such combinations are hard to exercise due to the vast interaction space. Guided by this study, we design CaMFuzz. CaMFuzz first builds a Configuration Model Specification through static analysis to extract configuration constraints, model profiles, and explicit configuration–model dependencies, generating valid yet diverse configurations with compatible model-side requirements. It then employs configuration–model interaction fuzzing that generates compatible initial models via templates and iteratively mutates model structures under train logic coverage guidance to explore implicit interactions. We evaluate CaMFuzz on four frameworks using a single-host server with eight GPUs. Within this experimentally validated setting, CaMFuzz achieves 75.5%–90.6% higher train logic coverage than representative baselines and detects 20 previously unknown DTBs. Inter-host networking and large-scale cluster behavior remain outside the experimentally validated scope of this work.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: Testing, Distributed Training Framework

ACM Reference Format:
Zhen Yan, Yuanliang Chen, Fuchen Ma, Ming Yan, Zehong Yu, Zhensheng Xian, and Yu Jiang. 2027. CaMFuzz: Testing Distributed Training Frameworks via Abundant Configuration–Model Interactions. In *22nd European Conference on Computer Systems (EuroSys '27), April 19–23, 2027, Rabat, Morocco.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3842654.3848560

1 Introduction

Distributed training frameworks are the core infrastructure supporting large-scale deep learning model training. They automatically manage computation partitioning, communication synchronization, and execution scheduling across distributed workers, ranging from multiple GPUs within a single host to workers deployed across multiple hosts [17, 28, 45, 46, 48]. In this work, we focus specifically on distributed training in the single-host, multi-GPU setting. In practice, users only need to configure training strategies, such as data parallelism, model parallelism, mixed precision, or their combinations, to train various deep learning models using distributed training frameworks, without manually handling low-level communication, synchronization, and scheduling details. As model scales and training costs continue to grow, distributed training frameworks have become indispensable components of modern deep learning infrastructure [10, 27, 39, 49].

Distributed training frameworks are complex, as they are designed to support efficient parallel training of large-scale models. They coordinate mechanisms such as distributed

*Yuanliang Chen and Yu Jiang are the corresponding authors.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
EuroSys '27, Rabat, Morocco
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2971-3/2027/04
https://doi.org/10.1145/3842654.3848560

gradient synchronization, pipeline scheduling, and mixed-precision computation, which can introduce subtle vulnerabilities. Once triggered, these issues often lead to severe consequences, including crashes during training or checkpointing, unresponsive hangs, and numerical errors or abnormal gradient propagation [34]. Such failures can interrupt training, degrade performance, or even render results invalid, resulting in substantial losses in time and computational resources for large-scale training tasks. We collectively refer to these issues as **Distributed Training Bugs** (DTBs).

To understand these bugs, we first conducted a comprehensive study on DTBs based on 133 real-world cases from four widely used distributed training frameworks, including DeepSpeed [39], Accelerate [10], ColossalAI [27], and Megatron-LM [49]. For each case, we manually analyzed its issue report and the corresponding fix to thoroughly understand its symptom severity and triggering conditions. Our study revealed two key insights: (1) The vast majority of DTBs (93%) require valid training configurations to trigger, where validity demands satisfying both intra-configuration constraints and explicit configuration–model dependencies (e.g., expert parallelism requires the model to contain MoE layers); without such validity, training fails at initialization and the bug-triggering execution path is never reached. (2) Beyond validity, nearly half of DTBs (45%) further require diverse configurations to interact with specific model structures (e.g., MoE layers, LoRA adapters, pipeline architectures) through implicit dependencies, since different configuration–model combinations activate different training execution paths that expose distinct bugs.

Researchers have proposed various techniques for detecting DTBs. Distributed training framework testing tools like D^3 [55] construct test cases for DTB detection but typically cover only a small subset of configuration parameters and test against fixed model types. General deep learning testing tools such as NNSmith [33] generate diverse models through graph-level mutations, but they target single-node frameworks and do not exercise distributed training configurations. However, many DTBs require coordinated interactions between specific training configurations and specific model structures to be triggered, a scenario existing tools struggle with due to their limited exploration of the configuration–model interaction space. Additionally, randomly generating configurations is ineffective, as constraints within configurations and dependencies between configurations and models cause many generated configuration–model pairs to be invalid or fail during initialization.

To effectively detect DTBs in distributed training frameworks, there are two main challenges: (1) **The first challenge** is to generate valid yet behaviorally diverse training configurations together with compatible model-side requirements. Distributed training frameworks expose large numbers of configuration parameters, but their validity is determined not only by intra-configuration constraints (e.g., CPU offloading requires compatible optimizer types) but also by explicit configuration–model dependencies (e.g., expert parallelism requires the model to contain MoE layers with expert routing). Violating either type of constraint produces invalid configuration–model pairs that fail during initialization, preventing effective testing. (2) **The second challenge** lies in efficiently exploring the model space under valid configuration–model compatibility constraints. Even after explicit dependencies are satisfied, different model structures can activate different execution paths under the same configuration, revealing implicit configuration–model interaction. The large number of possible configuration–model combinations makes it difficult to effectively explore this interaction space.

To address these challenges, we propose CaMFuzz, a comprehensive testing framework designed to automatically detect DTBs through systematic configuration–model interaction exploration. First, CaMFuzz introduces a Configuration Model Specification (CMSpec) that automatically extracts configuration constraints, model profiles, and explicit configuration–model dependencies from the framework's source code, enabling the generation of semantically valid yet behaviorally diverse training configurations with compatible model-side requirements. Second, to effectively explore the model space under these compatibility constraints and uncover bugs hidden in deep execution paths, CaMFuzz employs a configuration–model interaction fuzzing strategy. It first uses template-based generation to produce initial models that satisfy the model-side requirements derived from CMSpec, and then leverages train logic coverage data from the framework's core distributed training modules to guide model mutation during fuzzing. In this way, CaMFuzz conducts efficient configuration–model interaction testing with diverse and valid configuration–model pairs, effectively exercising distributed training mechanisms and detecting bugs across different distributed training frameworks.

We implemented CaMFuzz and evaluated it on four widely used distributed training frameworks: DeepSpeed, Accelerate, ColossalAI, and Megatron-LM, using a single-host server with eight GPUs. Our evaluation is limited to this single-host setting and does not validate the applicability of CaMFuzz to inter-host distributed training. In this setting, compared with state-of-the-art testing tools D^3 and NNSmith, CaMFuzz detected more DTBs and achieved 90.6% and 75.5% higher train logic coverage on average, respectively. Additionally, CaMFuzz identified 20 previously unknown DTBs in total, with 4 in DeepSpeed, 5 in Accelerate, 9 in ColossalAI, and 2 in Megatron-LM.

In summary, we make three key contributions:

- We conducted a detailed study of 133 real-world DTBs in four widely used distributed training frameworks and summarized their symptoms and triggering conditions.

- We designed CaMFuzz, a testing framework that automatically constructs a CMSpec to generate valid configurations, and employs configuration–model fuzzing to effectively detect DTBs.
- We implemented and evaluated CaMFuzz on four widely used distributed training frameworks using a single-host, eight-GPU setup. It detected 20 previously unknown DTBs, all confirmed and fixed by developers. We open-source CaMFuzz[1] for broader practical use.

2 Background

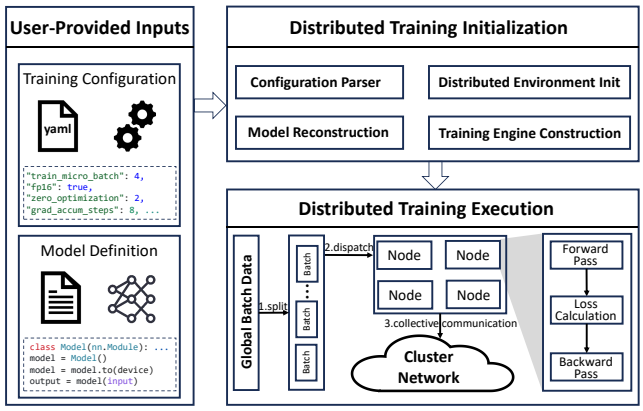


Figure 1. Overview of how distributed training frameworks coordinate training configurations and model structures across multiple nodes.

Distributed training is the standard approach for training large-scale deep learning models that exceed the capacity of a single device [9, 30]. As illustrated in Figure 1, distributed training frameworks accept two kinds of user-provided inputs, a training configuration and a model definition, and automatically manage the full distributed execution lifecycle. During initialization, the framework parses the configuration, sets up the distributed environment, restructures the model according to the specified parallelism strategy, and constructs the training engine. At runtime, it splits global batch data into micro-batches, dispatches them across nodes for forward and backward passes, and coordinates gradient synchronization through collective communication over the cluster network. Representative frameworks such as DeepSpeed, Accelerate, ColossalAI, and Megatron-LM provide rich support for data parallelism, tensor parallelism, pipeline parallelism, expert parallelism, ZeRO-based memory optimization, activation checkpointing, and mixed-precision training [3, 12, 13, 25, 38, 45].

Of the two kinds of inputs, **training configurations** specify the parameters that govern distributed execution, such as parallelism strategies, memory optimizations, numerical precision modes, and hyperparameters such as micro-batch size and gradient accumulation steps. These settings determine which internal execution paths of the framework are activated, affecting communication patterns, memory management, optimizer behavior, and synchronization mechanisms [31, 32, 41, 47, 69, 73, 74]. Complementing this, **model structures** define the neural network architecture to be trained, encompassing layer types, layer depth, hidden dimensions, and connectivity patterns. The model definition is restructured by the framework during initialization. For instance, it may be partitioned across pipeline stages or have its parameters sharded across data-parallel ranks. Because the training configuration and model structure jointly determine which execution paths are activated, neither can be considered in isolation [2, 11, 15, 18, 40, 53, 66]. A configuration that behaves correctly with one architecture may trigger a latent bug when paired with a different model structure. Errors that never appear under a single configuration or a fixed model often manifest only when particular configurations and model characteristics co-occur, leading to crashes, silent numerical corruption, or training hangs. Because these conditions require specific combinations of configuration parameters, model properties, and runtime interactions, existing testing techniques rarely explore this execution space, leaving configuration–model interaction bugs largely untested.

Throughout this paper, an explicit configuration–model dependency is a model requirement imposed by a training configuration and required for the configuration–model pair to successfully initialize and execute. An implicit configuration–model interaction refers to a valid configuration–model pair whose combined properties jointly activate a specific runtime execution path or behavior that would not be determined or exercised by either the training configuration or the model structure in isolation.

3 Motivation Study

To better understand DTBs in practice and guide our detection design, we conducted an empirical study on real-world cases. We selected four widely used distributed training frameworks: DeepSpeed, Accelerate, ColossalAI, and Megatron-LM. We then collected issue reports from their official GitHub trackers over the past two years, obtaining **2,639** issues in total. For each issue, we manually reviewed its description, reproduction steps, developer discussions, and related patches. We retained a case only when we could reproduce it using the provided steps or confirm the defect through source-code analysis supported by the discussion or patch. After filtering, we identified **133** confirmed DTB cases as shown in Table 1.

Table 1. Number of DTBs identified in each framework.

	DeepSpeed	Accelerate	ColossalAI	Megatron	Total
Total Issues	860	655	180	944	2639
DTBs	59	29	8	37	133

[1] CaMFuzz at: https://anonymous.4open.science/r/CaMFuzz-5623

3.1 Symptoms of DTBs

Finding 1: DTB symptoms are severe and diverse. Most DTBs (61.7%) cause training crashes, 28.6% lead to silent numerical anomalies, and 9.8% result in training hangs. 82 DTBs trigger runtime errors (e.g., `RuntimeError`, `AssertionError`), which stop training immediately and waste computational resources, especially in long-running or large-scale jobs. For example, DeepSpeed bug #7747 [51] causes a dtype mismatch crash under mixed-precision training. 38 DTBs produce numerical anomalies, such as incorrect gradients, abnormal loss values, or failed parameter updates. These issues are hard to detect because training appears normal while the model is corrupted. For instance, DeepSpeed bug #7640 [60] leads to integer overflow in FusedAdam, and Megatron-LM bug #1330 [52] causes duplicated loss accumulation. 13 DTBs cause training to hang without errors, making them waste GPU resources. For example, Megatron-LM bug #873 [44] leads to a deadlock due to incorrect communication ordering.

Table 2. Symptom classification of DTBs across four frameworks.

Symptom	DeepSpeed	Accelerate	ColossalAI	Megatron	Total (%)
Training Crash	37	21	4	20	82 (61.7%)
Numerical Error	18	6	1	13	38 (28.6%)
Training Hang	4	2	3	4	13 (9.8%)
Total	59	29	8	37	133 (100%)

3.2 Triggering Conditions of DTBs

Finding 2: The vast majority of DTBs (93%) require valid training configurations satisfying explicit configuration–model dependencies to be triggered. As shown in Table 3, 124 out of 133 DTBs (93%) can only be triggered when training proceeds with valid configurations; otherwise, training fails at initialization and the bug-triggering execution path is never reached. Validity here means that a configuration must satisfy both intra-configuration constraints (e.g., enabling CPU offloading requires compatible optimizer types) and explicit configuration–model dependencies (e.g., expert parallelism requires the model to contain MoE layers), so that training can proceed to the stage where bugs become reachable. The triggering configurations span multiple dimensions of distributed training, including parallelism strategies (DP, TP, PP, EP), memory optimizations (ZeRO stages, CPU offload, activation checkpointing), numerical precision (fp16, bf16, fp8), communication optimizations (gradient overlap, contiguous gradients), and checkpointing mechanisms (async checkpointing, distributed checkpoints). For instance, DeepSpeed bug #5203 [24] is triggered only when ZeRO-2 CPU offloading is enabled together with an MoE model under `ep_size > 1`, where the configuration's explicit dependency on MoE layers must be satisfied for the faulty expert gradient synchronization path to be reached. The case shows that DTBs are frequently hidden under valid configurations whose explicit dependencies on model structures are satisfied, and constraint-unaware configuration testing that ignores such dependencies is insufficient to expose them.

Table 3. Training configuration dependency of DTBs.

Dependency	DeepSpeed	Accelerate	ColossalAI	Megatron	Total (%)
Requires specific config	57	27	5	35	124 (93%)
No specific config needed	2	2	3	2	9 (7%)

Finding 3: Implicit configuration–model interactions play a central role in triggering DTBs. As shown in Table 4, DTBs fall into two categories based on their triggering conditions: Explicit only DTBs are triggerable under any model satisfying the configuration's explicit dependencies, while Explicit + Implicit DTBs additionally require specific model structures to activate the faulty execution path through implicit dependencies. Nearly half of DTBs (45%) fall into the latter category, where the same configuration activates different execution paths depending on the model's structural properties, exposing different bugs. Even when explicit dependencies are satisfied and training proceeds normally, specific model characteristics are still required to enter the faulty code paths. For example, Megatron-LM bug #1421 [5] is triggered only when MoE layers activate the token dispatcher path, and DeepSpeed bug #7632 [1] manifests only when LoRA layers invoke the framework's special handling for trainable parameters. These results indicate that triggering DTBs requires not only valid configurations but also specific model structures that activate the corresponding execution paths.

Table 4. Triggering conditions of DTBs categorized by configuration–model dependencies.

Triggering Condition	DeepSpeed	Accelerate	ColossalAI	Megatron-LM	Total	%
Explicit only	33	16	1	14	64	48%
Explicit + Implicit	24	11	4	21	60	45%
Neither	2	2	3	2	9	7%

3.3 A Motivating Example

DTBs caused by configuration–model interactions can lead to severe consequences in distributed training frameworks. One such example is a DTB in DeepSpeed (Issue #5203 [24]), where the interaction between ZeRO Stage-2 CPU offloading and a model containing MoE layers causes a device–backend mismatch during expert gradient norm synchronization, crashing training at every parameter update. Figure 2 illustrates the key steps to trigger this bug, and Figure 3 presents the core code snippet.

In DeepSpeed, the ZeRO Stage-2 optimizer partitions gradient and optimizer states across data-parallel ranks.

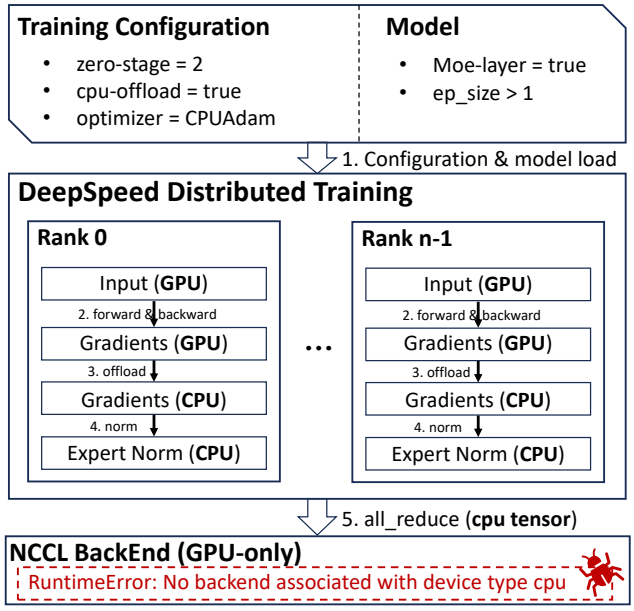


Figure 2. The DeepSpeed DTB causes a training crash when MoE expert gradient norms on CPU are passed to the GPU-only NCCL backend for all-reduce.

```
1  def _average_expert_grad_norms(self, ...):
2    for i, norm in enumerate(norm_groups):
3      if self.is_moe_param_group[i]:
4        scaled_norm_tensor = norm * 1.0 / \
5          dist.get_world_size(
6            group=self.real_dp_process_group[i])
7  +   if self.device == 'cpu':
8  +    scaled_norm_tensor = scaled_norm_tensor
9  +     .to(get_accelerator().current_device_name())
10       dist.all_reduce(scaled_norm_tensor,
11         group=self.real_dp_process_group[i])
12 -   norm_groups[i] = scaled_norm_tensor
13 +   norm_groups[i] = scaled_norm_tensor.to(self.device)
```

Figure 3. The core code snippet of DeepSpeed #5203, showing the device mismatch issue and its fix.

When CPU offloading is enabled, these states are migrated to CPU memory, and the `DeepSpeedCPUAdam` optimizer performs parameter updates on the CPU. For models containing MoE layers, DeepSpeed maintains separate expert parameter groups and synchronizes their gradient norms via `dist.all_reduce` in the method `_average_expert_grad_norms()`, as shown in Figure 3. The bug is triggered as follows. The user trains an MoE model with `ep_size > 1` under ZeRO Stage-2 CPU offloading and `DeepSpeedCPUAdam`. Forward and backward passes complete normally, but expert gradient norms are computed and stored on CPU due to offloading. At `optimizer.step()`, DeepSpeed enters `_average_expert_grad_norms()`, computes `scaled_norm_tensor` on the CPU (line 4), and passes it directly to `dist.all_reduce` (line 10). Because the default NCCL backend supports only GPU-resident tensors, this results in a `RuntimeError: No backend type associated with device type cpu`, causing the training to crash. The fix (lines 7–9 and 12–13) adds a device check: if `self.device` is `'cpu'`, the tensor is moved to GPU before all-reduce and back to CPU afterward via `.to(self.device)`. This DTB does not manifest when using MoE without CPU offloading, nor when enabling CPU offloading without MoE layers. It also remains hidden when `ep_size = 1`, since no cross-rank synchronization is needed. The crash only arises when all three conditions hold: CPU offloading is enabled, the model uses MoE layers, and `ep_size > 1`, leading to a CPU tensor being passed to the GPU-only NCCL backend.

This example highlights two key lessons aligned with our findings: 1) DTBs are reachable only when training proceeds with valid configurations satisfying explicit configuration–model dependencies. Here, ZeRO-2 CPU offloading must pair with `DeepSpeedCPUAdam`, and expert parallelism explicitly requires MoE layers; otherwise, training never reaches the faulty path. This motivates CaMFuzz's CMSpec to generate valid configurations with compatible model-side requirements. 2) Even when explicit dependencies are satisfied, different model structures activate different code paths through implicit dependencies. The expert gradient synchronization path is triggered only when the model contains MoE layers with `ep_size > 1`, a structural property not reflected in the configuration. This motivates CaMFuzz's configuration–model interaction fuzzing, which mutates model structures under coverage guidance to explore implicit dependencies.

3.4 Limitations of Existing Methods

Existing DTB detection techniques provide useful coverage for identifying training-time anomalies, but they fall short in thoroughly exercising the complex interactions between configurations and model structures. Two mainstream categories of methods dominate current practice, yet both fail to systematically explore the configuration–model interaction space where many DTBs arise.

Method 1: Limited configurations with fixed models. D^3 takes a systematic approach by constructing dedicated test cases for DTB detection. However, it typically covers only a small subset of configuration parameters (5–10 per framework) and tests against a fixed set of model types, relying on manual enumeration of configuration values and hand-crafted model definitions. According to Finding 2, 93% of DTBs require valid configurations satisfying explicit configuration–model dependencies, which manual enumeration with such restricted scope struggles to cover. Even worse, this restricted scope cannot effectively explore the implicit dependencies revealed by Finding 3: different model structures activate different framework code paths

under the same configuration, and bugs like the motivating example require specific model characteristics that fixed models do not provide. Consequently, the 45% of DTBs that depend on implicit configuration–model interactions remain largely undiscoverable.

Method 2: Diverse models without distributed configurations. NNSmith generates diverse neural network models through graph-level mutations and detects bugs via differential testing. While its model generation capability could in principle expose model-dependent DTBs, NNSmith targets single-node deep learning libraries and does not construct distributed training configurations such as parallelism strategies, ZeRO stages, or mixed-precision settings. As a result, the generated models are executed under default or trivial configurations that fail to activate the distributed training code paths where DTBs reside. According to Findings 2 and 3, neither valid configurations satisfying explicit dependencies nor the implicit configuration–model interactions that trigger 45% of DTBs can be exercised by model generation alone, leaving the configuration dimension of the interaction space entirely unexplored.

Key insight and technical challenge of CaMFuzz. The key challenge is not simply to diversify configurations and models independently, but to explore their joint space while preserving the semantic dependencies required for distributed training to execute. A generated configuration may violate intra-configuration constraints, while a diverse model may fail to satisfy structural requirements imposed by the configuration; in either case, execution terminates during initialization before meaningful training logic can be exercised. Even after these explicit dependencies are satisfied, different compatible model structures can activate substantially different runtime paths under the same configuration, making implicit configuration–model interactions difficult to explore efficiently.

CaMFuzz addresses this challenge by tightly coupling CMSpec-guided configuration generation with dependency-preserving, coverage-guided model exploration. CMSpec extracts configuration constraints, model profiles, and explicit configuration–model dependencies, and generates each training configuration together with its model-side requirements. For each generated configuration, CaMFuzz instantiates a compatible model template and mutates model structures while continuously preserving these requirements. Train logic coverage then guides the mutation process toward previously unexplored distributed-training execution paths, enabling the search to expose implicit configuration–model interactions without sacrificing validity. Thus, CMSpec and interaction fuzzing form a closed-loop exploration process rather than two independent testing stages, allowing CaMFuzz to reach DTBs that isolated configuration or model exploration is likely to miss.

4 CaMFuzz Design

4.1 CaMFuzz Workflow

Figure 4 illustrates the workflow of CaMFuzz, which consists of two main phases. The first phase focuses on Configuration Model Specification Construction: (1) The CMSpec Constructor analyzes the target framework's source code. (2) It extracts training configuration constraints and model profiles from the code. (3) These are used to build the CMSpec. (4) The CMSpec generates valid training configurations that satisfy the extracted constraints.

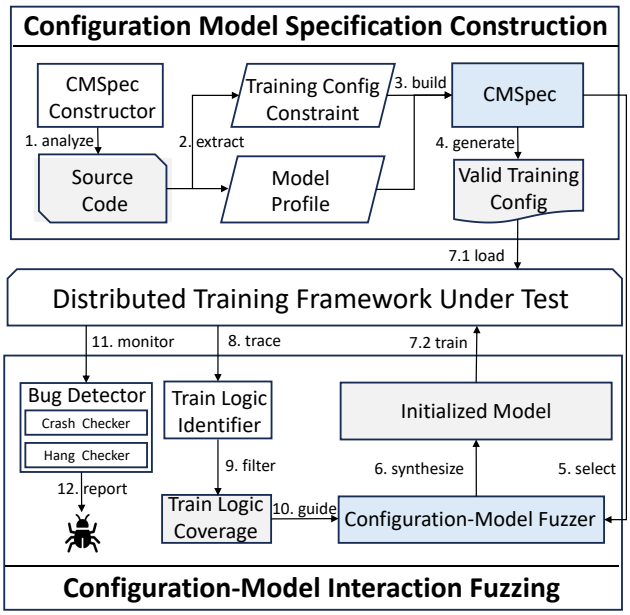


Figure 4. Workflow of CaMFuzz. It includes two main phases: (1) Configuration Model Specification Construction and (2) Configuration–Model Interaction Fuzzing.

Based on the generated configuration c, CaMFuzz proceeds to the second phase, Configuration–Model Interaction Fuzzing, consisting of the following steps: (5) A compatible model template is selected based on the structural requirements imposed by c. (6) The Configuration–Model Fuzzer synthesizes a concrete model instance from the selected template. (7) The generated configuration and synthesized model are loaded into the distributed training framework under test (7.1), which then executes the training process (7.2). (8) A Train Logic Identifier traces the execution of the framework's core distributed training modules. (9) The trace is filtered to compute train logic coverage over training-related code. (10) The coverage feedback guides subsequent fuzzing iterations, steering model mutations toward unexplored execution paths. (11) The Bug Detector monitors execution outcomes, checking for crashes and hangs. (12) Detected anomalies are reported as distributed training bugs (DTBs).

Configuration–model pairs that lead to new train logic coverage are prioritized by the Fuzzer to guide the generation of subsequent model mutations. CaMFuzz iteratively repeats steps (5)–(12), mutating model structures under coverage guidance while keeping the configuration fixed, until the time budget is exhausted.

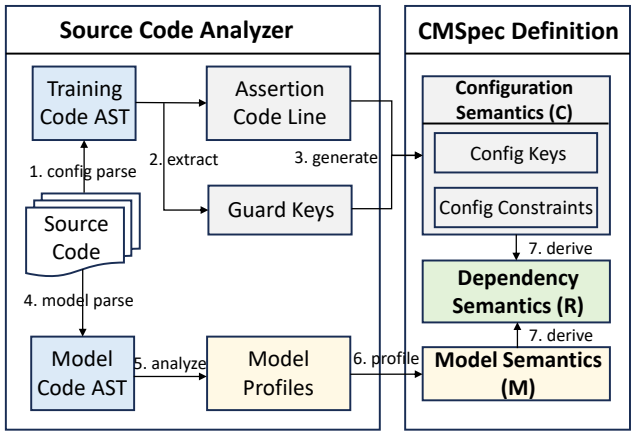


Figure 5. Process of CMSpec construction via static analysis of framework source code.

4.2 CMSpec Construction

In distributed training, training configurations are subject not only to their own constraints but also to explicit dependencies between configurations and models. It is therefore necessary to understand which models are compatible with a given configuration. To this end, CaMFuzz constructs a Configuration Model Specification (CMSpec) to capture three types of semantics specific to distributed training frameworks: training configuration semantics, model semantics, and the dependency semantics between them.

Semantic Extraction. Formally, we define CMSpec as $\mathcal{T} = (C, M, R)$, where C denotes the training-configuration semantics, M denotes the model semantics, and R captures the dependency relations between them. As shown in Figure 5, CaMFuzz parses the framework source code into an abstract syntax tree (AST) and analyzes both distributed-training-related code and model-related code.

To construct C, CaMFuzz analyzes `assert` and `raise` statements in distributed training components to extract configuration constraint semantics, i.e., conditions under which configuration parameters must satisfy certain relations or may lead to conflicts. It further identifies configuration keys through patterns such as constants, dictionary accesses, and `self`-attribute chains. When a constraint is guarded by conditional branches (e.g., `if` statements), CaMFuzz extracts the corresponding guard keys, which determine when the constraint becomes active. These guard-aware constraints capture the conditional semantics of training strategies.

To construct M, CaMFuzz performs AST-level analysis on model-related code to extract model semantics in the form of a model profile. This includes structural properties relevant to distributed execution, such as model types, layer organization, parameter shapes, and the presence of specific architectural components (e.g., pipeline modules, MoE routing, or Transformer blocks). These semantics characterize the space of models that can meaningfully interact with distributed training strategies.

To construct R, CaMFuzz establishes dependency relations between C and M by matching configuration semantics with model-side requirements implied by the code. Specifically, it analyzes where training strategies impose structural assumptions and constraints on models, such as requiring partitionable modules for pipeline parallelism, dimensional divisibility for tensor parallelism, or expert-related components for expert parallelism. As a result, R encodes the explicit dependencies that determine which model structures are compatible with a given configuration and can activate the corresponding distributed training logic.

Semantics-Aware Configuration Generation. Guided by CMSpec, CaMFuzz generates configurations in a top-down manner. It first selects a distributed training strategy (e.g., DP, TP, PP, EP, or their combinations) and determines a valid topology under hardware constraints. It then samples concrete parameter values and dependency conditions extracted into C. Constraints are processed by increasing guard-key depth: unconditional constraints are handled first, followed by those activated by previously chosen strategy options. During this process, CaMFuzz simultaneously derives the corresponding model-side requirements from R. Therefore, CMSpec outputs a configuration along with its corresponding model-side requirements. Constraints that cannot be resolved statically are checked by the target framework during initialization. Configuration–model pairs rejected before the training loop are discarded and are not inserted into the seed pool.

4.3 Configuration–Model Interaction Fuzzing

Given a valid training configuration c, CaMFuzz fuzzes the interaction between the configuration and model structures to uncover DTBs hidden in diverse execution paths. To achieve this, CaMFuzz first constructs an initial model that satisfies all explicit dependencies imposed by c, ensuring that the pair can successfully enter the training loop. It then iteratively mutates the model structure under training code coverage guidance, exploring implicit dependencies between configurations and models to uncover deeper hidden DTBs.

4.3.1 Initial Model Generation. CaMFuzz maintains a pool of model templates collected from two sources: official training example scripts provided by various frameworks, and parameterized training scripts developed by us that incorporate DTB-related behaviors identified in our empirical

study (e.g., checkpoint saving followed by loading within a single run). For each template, CaMFuzz extracts a model profile expressed in M using the same model-related code analysis performed during CMSpec construction.

Given a training configuration c and the dependency relations R from CMSpec, CaMFuzz first derives an explicit set of structural requirements $Req(c)$ by querying model-side constraints associated with the configuration options in R. For example, if pipeline parallelism is enabled in c, R specifies that the model must be partitionable. Next, CaMFuzz matches $Req(c)$ against the model profiles of all templates and filters those whose structural properties satisfy the dependencies. One template is then selected uniformly at random from the candidates and instantiated by assigning concrete values to its configurable parameters, ensuring that all structural and dimensional constraints encoded in $Req(c)$ are satisfied. This process produces an initial configuration–model pair (c, m_0) that can successfully initialize and enter the training loop.

4.3.2 Coverage-Guided Model Mutation.

Although the initial model (m_0) satisfies all explicit dependencies and can be trained successfully, it represents only a single point in a large model space. To explore implicit dependencies between training configurations and model structures and to reach deeper execution paths, CaMFuzz mutates the model structure while preserving explicit dependencies, using the framework's train logic coverage as feedback to guide mutations toward previously unexplored regions.

Train Logic Coverage Collection. CaMFuzz collects full line coverage on each target framework's source code during distributed executions, and then post-processes the merged reports to extract only the coverage related to distributed training. Specifically, it retains only items whose source locations fall within distributed-training code regions, such as launcher and communication paths, ZeRO and sharding logic, major parallelism mechanisms, and related runtime or group-management code. This filtered coverage, referred to as *train logic coverage*, focuses the feedback signal on the framework components most relevant to DTBs, so that mutations are guided toward execution paths within distributed training logic rather than unrelated utility or framework-wide code.

Mutation operators. Guided by our empirical study, we design three mutation operators, each targeting a model-level characteristic identified as frequently involved in triggering real-world DTBs:

- *Layer parameter mutation.* Modifies numerical parameters of existing layers (e.g., hidden size, number of attention heads, intermediate dimension) within the valid ranges imposed by the configuration. This operator targets the parameter-property dependencies identified in Finding 3: for instance, DeepSpeed bug #7640 [60] is triggered only when a parameter tensor exceeds 2^{31} elements, a condition determined entirely by layer dimensions.
- *Layer count mutation.* Adds or removes layers while preserving configuration constraints. Model depth affects pipeline partitioning, activation checkpointing, and memory management. For example, Megatron-LM bug #873 [44] appears only under specific pipeline partitions induced by certain layer counts.
- *Parameter freeze mutation.* Toggles the `requires_grad` attribute of selected parameters or layers to simulate fine-tuning and LoRA-style training. Mixed trainable and frozen parameters can interact with gradient synchronization and optimizer state management in non-trivial ways. For example, DeepSpeed bug #7632 [1] occurs only with partially frozen LoRA adapters.

Algorithm 1 Configuration–Model Interaction Fuzzing

Require: CMSpec $\mathcal{T}$, framework F, time budget T
Ensure: Detected DTBs $\mathcal{B}$

1: $\mathcal{B} \leftarrow \emptyset$; $\mathcal{S} \leftarrow \emptyset$; $covMap \leftarrow \emptyset$
2: **while** elapsed time $< T$ **do**
3: **if** $\mathcal{S} = \emptyset$ **or** with probability p_{new} **then**
4: $c, m_0 \leftarrow \textsc{GenerateConfig}(\mathcal{T}), \textsc{GenerateModel}(c)$
5: $(c, m) \leftarrow (c, m_0)$
6: **else**
7: $(c, m) \leftarrow \textsc{SelectSeed}(\mathcal{S})$
8: **end if**
9: $m' \leftarrow \textsc{Mutate}(m)$
10: **if** $m' \not\models Req(c)$ **then**
11: **continue**
12: **end if**
13: $result, cov \leftarrow \textsc{Execute}(F, c, m')$
14: **if** $result$ is DTB **then**
15: $\mathcal{B} \leftarrow \mathcal{B} \cup \{(c, m')\}$
16: **end if**
17: **if** $cov \setminus covMap \neq \emptyset$ **then**
18: $covMap, \mathcal{S} \leftarrow covMap \cup cov, \mathcal{S} \cup \{(c, m')\}$
19: **end if**
20: **end while**
21: **return** $\mathcal{B}$

Fuzzing loop. Algorithm 1 outlines the overall testing process. CaMFuzz first initializes a seed pool of configuration–model pairs (line 1). During each iteration (line 2), it either generates a new pair (c, m_0) from CMSpec via template-based generation (lines 4–5), or selects an existing seed from the pool (line 7), thereby balancing exploration and reuse. Given a selected pair, the configuration is kept fixed while the model is mutated to produce a new candidate (line 9). If the mutated model violates $Req(c)$, the iteration is skipped (lines 10–12); otherwise, the pair is executed to obtain the result and coverage (line 13). If a DTB is triggered, the pair is recorded (lines 14–16). Meanwhile, if new coverage is observed (line 17), the pair is added back to the seed pool (line 18), guiding the search toward unexplored paths.

5 Implementation

We implemented CaMFuzz and evaluated it across four widely used distributed training frameworks: DeepSpeed v0.14, Accelerate v0.33, ColossalAI v0.3.7, and Megatron-LM v0.8. These frameworks differ in their levels of abstraction, parallelism strategies, and target use cases, collectively covering a broad spectrum of distributed training scenarios. DeepSpeed provides ZeRO-based memory optimization and supports advanced features such as CPU offloading and mixed-precision training. Accelerate offers a lightweight, high-level API for distributed training and is widely adopted within the open-source NLP community. ColossalAI enables hybrid parallelism, including data, tensor, and pipeline parallelism. Megatron-LM is designed for high-performance training of large-scale Transformer models, leveraging sophisticated parallelism techniques.

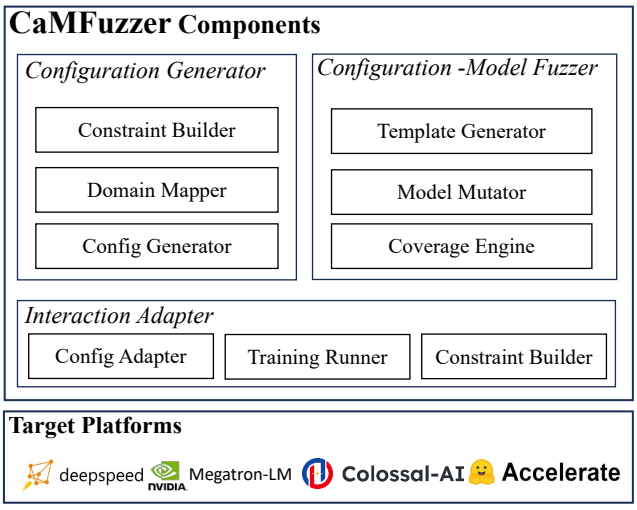


Figure 6. CaMFuzz components: Configuration Generator, Configuration–Model Fuzzer, and Interaction Adapter.

Figure 6 illustrates the components of CaMFuzz, which are divided into three main parts. The first part is the **Configuration Generator**, which consists of three components. First, the Constraint Builder parses the target framework's source code into an Abstract Syntax Tree and extracts configuration constraints and explicit configuration–model dependencies from `assert` statements, `raise` statements, and conditional branches. Second, the Domain Mapper infers value domains from default values and configuration declarations. Finally, the Config Generator synthesizes configurations that satisfy the extracted constraints and derives the corresponding model-side requirements. The second part is the **Configuration–Model Fuzzer**, which selects and instantiates compatible model templates given a configuration via the Template Generator, applies the three mutation operators via the Model Mutator, and instruments the framework's core distributed training modules to collect train logic coverage and guide subsequent mutations via the Coverage Engine. The third component is the **Interaction Adapter**, which connects CaMFuzz to a target framework by handling configuration translation, training execution, and framework- or hardware-specific constraints. This design isolates framework-dependent logic and keeps CaMFuzz largely framework-agnostic.

Automation and Implementation Basis. The source analyses used by the Constraint Builder and model profiler are custom-built on Python's standard `ast` module. CaMFuzz automatically performs source analysis, configuration generation, model profiling, model mutation, execution, coverage collection, seed scheduling, and crash/hang detection. Runtime-only constraints are checked during framework initialization, and configuration–model pairs that fail initialization are discarded automatically.

Adaptation to a New Distributed Training Framework. Porting CaMFuzz leaves the core generation and fuzzing algorithms unchanged, but requires four framework-specific tasks: (1) translating CaMFuzz's internal configuration representation into the framework's configuration format; (2) integrating the framework launcher and training entry point into the Training Runner; (3) identifying the distributed-training modules included in train logic coverage; and (4) constructing parameterized templates from framework APIs and official training examples. Across the four evaluated frameworks, we constructed 42 templates: 8 for DeepSpeed, 12 for Accelerate, 18 for ColossalAI, and 4 for Megatron-LM. The complete adaptation required an average of 19 person-hours per framework.

6 Evaluation

To evaluate the effectiveness of CaMFuzz, we compare it against two representative baselines: D^3 [55], an equivalence-based DTB testing tool, and Adapted-NNSmith, our distributed-training adaptation of NNSmith [33]. We applied all tools to four mainstream distributed training frameworks: DeepSpeed v0.14, Accelerate v0.33, ColossalAI v0.3.7, and Megatron-LM v0.8. All experiments were conducted on an Ubuntu 22.04.3 LTS system with Linux kernel version 5.15.0, deployed on a single server equipped with 8 NVIDIA Tesla V100S-PCIE GPUs, dual AMD EPYC 7763 processors providing 128 physical CPU cores, and 1 TiB of main memory. The system uses NVIDIA driver 535.216.03 and PyTorch 2.8.0 with CUDA 12.8 support. The 8 GPUs are distributed across two NUMA nodes and connected through intra-host PCIe interconnects. This is a single-host, multi-GPU environment: all distributed workers execute on one server. The experiments therefore evaluate framework-level configuration–model interactions under multi-GPU execution, but do not exercise inter-host networking, cross-machine launchers, network topology, or scale-dependent behavior of large clusters. All experiments were conducted

for 48 hours and repeated three times under identical conditions, and the average results are used in this paper. Based on this experimental setup, we designed experiments to address the following four research questions:

- **RQ1:** Is CaMFuzz effective in detecting DTBs in real-world distributed training frameworks?
- **RQ2:** Can CaMFuzz cover more framework code compared with state-of-the-art methods?
- **RQ3:** Do the CMSpec and coverage-guided model fuzzing effectively improve testing performance?
- **RQ4:** How accurately and completely does CMSpec capture the constraints needed for effective testing?

6.1 Bugs in Real-World Distributed Training Systems

We applied CaMFuzz to all four target frameworks and compared it with two representative baselines under the same setup. D^3 detects discrepancies by constructing equivalent distributed and single-node executions, while its input space uses a manually enumerated subset of configuration parameters and fixed model types. We use *Original NNSmith* to denote the upstream system, which generates models for single-node differential testing and lacks distributed-training configurations and framework-specific launchers. To construct *Adapted-NNSmith*, we retain NNSmith's original model-generation strategy and add CMSpec-generated distributed configurations together with dedicated runners for DeepSpeed, Accelerate, ColossalAI, and Megatron-LM. Original NNSmith cannot directly execute these distributed workflows, so the quantitative tables report Adapted-NNSmith. Adapted-NNSmith does not use CaMFuzz's compatible template selection, dependency-preserving coverage-guided model mutation, or train-logic-based seed scheduling. It therefore provides diverse models under valid distributed configurations without implementing CaMFuzz's joint interaction exploration. Each experiment ran for 48 hours on each target.

As shown in Table 5, CaMFuzz uncovered 20 previously unknown DTBs across four frameworks: 4 in DeepSpeed, 5 in Accelerate, 9 in ColossalAI, and 2 in Megatron-LM. All 20 reports were confirmed and fixed by developers. For practical validation, we applied appropriately mutated versions of the corresponding generated configurations to official training scripts and established models provided by the respective frameworks, successfully reproducing all 20 bugs. Therefore, these reports do not depend solely on synthetic template programs. Most DTBs (19/20) lead to training crashes, typically due to errors in distributed computation such as device–backend mismatches, invalid tensor operations, incorrect reshaping, or synchronization failures. These crashes are especially costly in large-scale training, where runs can span weeks and consume millions of GPU-hours. Crashes during checkpointing (e.g., Bugs #12 and #15) may further corrupt saved states, forcing training to restart from scratch. The remaining bug causes training to hang indefinitely, blocking GPU resources without errors. Such hangs are harder to detect and recover from. For example, Bug #13 induces a deadlock under a combination of a learning rate scheduler, fp16, and ZeRO-1, leaving all distributed workers unresponsive until manual intervention.

Comparison with existing methods. In our 48-hour experiments, D^3 detected only 1 DTB (#10). Adapted-NNSmith found 7 bugs (#1–3, 5, 7, 8, 14). However, these tools were unable to uncover the remaining DTBs because they cannot systematically explore the configuration–model interaction space. Triggering these bugs requires specific combinations of training configurations and model structures, often involving subtle inter-parameter dependencies and implicit structural constraints. D^3 relies on manually enumerated configuration values and fixed model definitions, preventing it from reaching execution paths activated by non-default or multi-dimensional configuration combinations. Adapted-NNSmith generates diverse models under valid distributed configurations, but its configuration-independent model generation does not systematically preserve and explore configuration-specific structural requirements, limiting the paths it reaches. In contrast, CaMFuzz constructs a CMSpec through static analysis to generate valid yet diverse configurations, and employs coverage-guided model mutation to systematically explore implicit configuration–model interactions that other tools fail to cover. For instance, Bug #11 requires both a specific configuration and a specific model structure to be triggered: it arises when ZeRO-1 is used together with bf16 communication, and the model contains parameter tensors large enough to exceed the gradient reduction bucket size. This approach allows CaMFuzz to detect all 20 DTBs, demonstrating its effectiveness in identifying distributed training bugs in real-world frameworks and answering **RQ1**.

6.1.1 Case Study.

We present a representative case to illustrate how a DTB arises in a real distributed training framework and how CaMFuzz identifies it. This case corresponds to Bug #11 in Table 5, discovered on DeepSpeed. The bug occurs in the ZeRO Stage-1 gradient reduction logic, where extra-large parameters are recorded under `param.dtype` but later looked up using `comm_dtype`, causing a key mismatch that leads to erroneous gradient reduction across data-parallel workers when `communication_data_type` is set to `bf16`. As a result, training crashes with a `RuntimeError`. The code snippet in Figure 7 describes this DTB and its fix.

Root Cause. In DeepSpeed's ZeRO Stage-1 optimizer, when a parameter's element count exceeds the configured bucket size, it is recorded in a separate dictionary for extra-large gradient reduction. The original code stores this parameter under the parameter's own data type (line 12) but looks it up using the communication data type (line 4). When bf16 communication is enabled while the model uses fp32 parameters, the communication type resolves to bfloat16 whereas

Table 5. 20 previously unknown DTBs detected by CaMFuzz within 48 hours. All were confirmed and fixed by developers and reproduced using established models and official training workflows. Bug identifiers remain partially anonymized for the review process.

#	Platform	Root Cause	Identifier	Status
1	ColossalAI	CPU-offloaded tensors cannot be handled by CUDA-only `FusedLAMB` kernels when `initial_scale` is too small.	Bug#xxx1	Fixed
2	ColossalAI	CPU tensors cannot be correctly reduced in distributed `Lamb` under ZeRO CPU offload configuration.	Bug#xxx0	Fixed
3	ColossalAI	bitsandbytes optimizer assumes GPU-resident states, but ZeRO maintains optimizer states on CPU memory.	Bug#xxx9	Fixed
4	ColossalAI	Adagrad state entry `sum` incorrectly initialized under `LowLevelZeroPlugin`, causing inconsistent updates.	Bug#xxx6	Fixed
5	ColossalAI	FP8 matmul path cannot enforce requirement that input matrix dimensions must be divisible by 16.	Bug#xxx0	Fixed
6	ColossalAI	Optimized attention cannot handle tensor input for `attention_mask`, although a mapping structure is required.	Bug#xxx9	Fixed
7	ColossalAI	Optimizer offloading leaves tensors on CPU, but `FusedAdam` still invokes CUDA-specific fused operations.	Bug#xxx8	Fixed
8	ColossalAI	FP8 linear wrapper cannot correctly restore output tensor shape, leading to invalid reshape operations.	Bug#xxx7	Fixed
9	ColossalAI	Gemini forwards non-default `div_scale` to `CPUAdam`, violating its assumption of fixed scaling behavior.	Bug#xxx6	Fixed
10	DeepSpeed	ZeRO-3 quantized weights fail to keep bf16 precision, casting tensors to fp16 during training.	Bug#xxx5	Fixed
11	DeepSpeed	ZeRO-1 computes bf16 communication bucket ranges incorrectly, causing erroneous gradient reduction.	Bug#xxx3	Fixed
12	DeepSpeed	Elastic checkpointing mishandles zero-dimensional optimizer states, slicing them incorrectly as vector tensors.	Bug#xxx0	Fixed
13	DeepSpeed	A scheduler combined with fp16 and ZeRO-1 cannot maintain synchronization, leading to deadlock.	Bug#xxx8	Fixed
14	Accelerate	FSDP-sharded Adagrad states and gradients cannot stay shape-compatible during foreach updates.	Bug#xxx9	Fixed
15	Accelerate	FSDP2 invokes `optimizer.state_dict()` before Adagrad remapping stabilizes, causing inconsistent states.	Bug#xxx5	Fixed
16	Accelerate	Missing `fsdp_min_num_params` cannot be parsed correctly, as float-form text later fails integer conversion.	Bug#xxx4	Fixed
17	Accelerate	`NO_WRAP` with activation checkpointing cannot correctly set the auto-wrap policy, although it is still invoked.	Bug#xxx2	Fixed
18	Accelerate	Some optimizer parameters cannot be switched to their FSDP2 sharded counterparts under combined parallelism.	Bug#xxx1	Fixed
19	Megatron-LM	Overlapped gradient reduction cannot avoid duplicate registration of gradients in DDP bucket tracking.	Bug#xxx7	Fixed
20	Megatron-LM	Uniform recompute partitioning cannot bound Transformer layer indices when layer counts are not divisible.	Bug#xxx4	Fixed

Table 6. Bugs found by each tool within 48 hours.

	CaMFuzz	D^3	Adapted-NNSmith
Number of Bugs	20	1	7
Bug IDs	#1–20	#10	#1–3, 5, 7, 8, 14

```
def reduce_independent_p_g_buckets_and_remove_grads(
self, param, i):
  grad_reduc = self.get_gradient_for_reduction(param)
- bucket = self.ipg_buckets[self.get_param_comm_dtype(param)]
+ comm_dtype = self.get_param_comm_dtype(param)
+ bucket = self.ipg_buckets[comm_dtype]
  if bucket.elements + param.numel() > self.reduce_bucket_size:
    self.reduce_ipg_grads()
  ...
  if self.contiguous_gradients:
    if param.numel() > self.reduce_bucket_size:
-   self.extra_large_param_to_reduce[param.dtype] = param
+   self.extra_large_param_to_reduce[comm_dtype] = param
```

Figure 7. A ZeRO-1 bug in DeepSpeed where using param.dtype instead of comm_dtype breaks bf16 gradient reduction.

the parameter type remains float32, causing a key mismatch that crashes all workers with a runtime error during gradient synchronization. The fix consistently uses the communication type as the dictionary key. This bug exhibits an implicit dependency between configuration and model structure. The faulty path is entered only when a parameter's element count exceeds the bucket size, so triggering the bug requires both bf16 communication and a model with sufficiently large parameter tensors relative to the configured bucket size. D^3 neither includes bf16 communication in its configuration space nor calibrates model sizes against the bucket size. Adapted-NNSmith can execute a CMSpec-generated ZeRO configuration, but its original model-generation strategy does not use coverage feedback to calibrate parameter sizes against the reduction bucket. In contrast, CaMFuzz uses the CMSpec to generate a valid ZeRO Stage-1 configuration with bf16 communication. It first produces an initial model via template-based generation. It then applies the layer parameter mutation operator to increase layer dimensions. As a result, parameter tensors exceed the bucket threshold. This reliably enters the faulty extra-large parameter path and triggers the DTB.

Table 7. Historical DTBs reproduced by CaMFuzz.

DeepSpeed	Accelerate	ColossalAI	Megatron-LM	Total
53/59	26/29	7/8	34/37	120/133

Historical Failures Evaluation. We also evaluated CaMFuzz against the 133 historical DTB cases identified in our empirical study, configuring each framework at the historical version referenced in the corresponding issue report and running for 48 hours per framework. Since 40 of the 133 DTBs exhibit silent anomalies, we designed a specialized oracle for each based on its documented symptom and integrated them into CaMFuzz's Bug Detector. As shown in

Table 7, CaMFuzz successfully reproduced 120 of 133 historical DTBs (90.2%), including 37 of 40 silent numerical anomalies (92.5%). The high reproduction rate on silent anomalies confirms that CaMFuzz's configuration–model interaction exploration can effectively trigger execution paths where silent errors arise, demonstrating strong capability in both triggering and detecting silent DTBs when equipped with appropriate oracles. The 13 unreproduced DTBs are caused by factors orthogonal to configuration–model interactions: 6 require inter-host execution or large-scale GPU clusters beyond our single-host, 8-GPU environment; 3 depend on external environment conditions such as specific system libraries or container runtimes; 2 involve specific API usage patterns rather than configuration–model combinations; and 2 require special data or compilation conditions outside CaMFuzz's current scope.

6.2 Effectiveness on Train Logic Coverage

Table 8. Train–logic line coverage on four target systems in 48 hours. CaMFuzz consistently achieves the highest coverage.

Tool	DeepSpeed	Accelerate	ColossalAI	Megatron-LM	Improvement
Adapted-NNSmith	4,787	3,137	5,719	3,029	**↑75.5%**
D^3	1,748	3,161	4,928	5,506	**↑90.6%**
CaMFuzz	**5,942**	**3,719**	**10,472**	**9,118**	–

Since higher coverage generally increases the likelihood of finding bugs, we also evaluated CaMFuzz's ability to cover the training logic of distributed training frameworks. In conclusion, CaMFuzz consistently outperforms existing state-of-the-art tools across all four target distributed training frameworks. On average, it achieves 90.6% and 75.5% higher coverage of training logic code compared to D^3 and Adapted-NNSmith, respectively. The statistics adequately answer **RQ2**.

The key reason is that CaMFuzz explores both the configuration space and the model space, enabling broader test scenarios and activating more execution paths. Among baseline tools, D^3 has lower coverage because it uses limited configuration parameters and fixed models, failing to cover paths related to parallelism strategies, memory optimizations, and other advanced features. Adapted-NNSmith retains NNSmith's generic model-generation strategy and does not use configuration-specific template selection or dependency-preserving, coverage-guided mutation. As a result, the execution paths remain shallow, particularly in Megatron-LM. In contrast, CaMFuzz models configuration constraints and jointly explores configuration–model combinations, triggering more execution paths and achieving deeper, more comprehensive training logic coverage.

6.3 Ablation Study

To evaluate the contributions of the CMSpec and the configuration–model interaction fuzzing strategy, we conducted an ablation study comparing three CaMFuzz variants: (1) CaMFuzz, the full version with both CMSpec and configuration–model interaction fuzzing enabled; (2) CaMFuzz $\bar{}_C$, a variant that disables CMSpec and instead generates training configurations randomly, without enforcing the extracted parameter constraints and dependencies; and (3) CaMFuzz $\bar{}_M$, a variant that retains CMSpec but disables the configuration–model interaction fuzzing strategy, using random model generation and mutation instead. We measured both the number of DTBs detected and the train logic code coverage achieved by each variant during the 48-hour testing period on all four frameworks.

Table 9. Comparison of CaMFuzz $\bar{}_C$, CaMFuzz $\bar{}_M$, and CaMFuzz in terms of bug detection and train logic coverage.

Framework	# Bugs			Coverage		
	CaMFuzz $\bar{}_C$	CaMFuzz $\bar{}_M$	CaMFuzz	CaMFuzz $\bar{}_C$	CaMFuzz $\bar{}_M$	CaMFuzz
DeepSpeed	1	2	4	5,143	5,288	5,942
Accelerate	2	3	5	3,256	3,335	3,719
ColossalAI	1	5	9	9,524	8,958	10,472
Megatron-LM	0	1	2	5,369	7,603	9,118
Total	4	11	20	23,292	25,184	29,251
Improvement	+400%	+82%	–	+25.6%	+16.1%	–

As shown in Table 9, with the support of both CMSpec and configuration–model interaction fuzzing, CaMFuzz detects all 20 DTBs and achieves the highest train logic coverage within 48 hours. In contrast, CaMFuzz $\bar{}_C$ detects only 4 DTBs, while CaMFuzz $\bar{}_M$ detects 11. The remaining DTBs require precise combinations of valid training configurations and specific model structures, which are difficult to exercise through random configuration generation or random model generation and mutation. In terms of train logic coverage, CaMFuzz $\bar{}_C$ achieves a total of 23,292 covered lines across four frameworks, while CaMFuzz $\bar{}_M$ achieves 25,184. CaMFuzz achieves 29,251, representing an improvement of 25.6% over CaMFuzz $\bar{}_C$ and 16.1% over CaMFuzz $\bar{}_M$. These results demonstrate the effectiveness of both components. Compared to CaMFuzz $\bar{}_C$, the CMSpec substantially enhances both bug detection (+400%) and train logic coverage by ensuring that the vast majority of generated configurations are valid and can proceed to meaningful training execution, rather than failing at initialization due to constraint violations. Similarly, compared to CaMFuzz $\bar{}_M$, the configuration–model interaction fuzzing strategy effectively improves both bug detection (+82%) and coverage by exploring diverse model structures and implicit configuration–model interactions. Across the three variants, greater total train logic coverage coincides with more detected bugs: 23,292 lines and 4 bugs for CaMFuzz $\bar{}_C$, 25,184 lines and 11 bugs for CaMFuzz $\bar{}_M$, and 29,251 lines and 20 bugs for the full system. This observation does not by itself establish causality, but it supports train logic

coverage as a useful search signal. These statistics provide a comprehensive answer to **RQ3**.

6.4 Accuracy and Completeness of CMSpec

Table 10. Accuracy of CMSpec configuration generation across four frameworks.

	DeepSpeed	Accelerate	ColossalAI	Megatron-LM	Average
Random (%)	29.2	2.9	56.9	0.4	22.4
CMSpec (%)	94.0	96.0	93.9	94.2	94.5

To evaluate the accuracy of CaMFuzz, we analyzed the validity of configurations generated by CMSpec. Specifically, we conducted 1,000 rounds of testing and repeated the experiment three times to examine whether the configurations generated by CMSpec were valid. The results are shown in Table 10. Random configuration generation achieves an average validity rate of only 22.4%, meaning that approximately three out of four generated configurations fail during framework initialization due to constraint violations. In contrast, CMSpec-guided generation achieves an average validity rate of 94.5%, representing a substantial improvement. The remaining 5.5% of invalid test cases are mainly caused by constraints that cannot be captured by static analysis alone, such as those expressed through factory patterns and dynamic dispatch. We use this successful initialization and execution rate as the accuracy measure for CMSpec. As a complementary system-level proxy for completeness, the historical-bug experiment in Table 7 shows that CaMFuzz reproduced 120 of 133 DTBs (90.2%); the 13 unreproduced cases depend on inter-host scale, external environments, API usage, or special data and compilation conditions outside the modeled configuration–model interaction space. Together, these results answer **RQ4**.

7 Discussion

Evaluation Scope. All experiments are conducted on a single server with eight GPUs connected through PCIe. Our evaluation therefore validates framework-level bugs arising from interactions between distributed-training configurations and model structures under single-host, multi-GPU execution within the scope of our current experimental setup. In particular, the evaluated setting covers bugs related to distributed execution mechanisms such as parallelism strategies, parameter sharding, gradient synchronization, mixed-precision training, and checkpointing when these mechanisms are exercised within a single host.

It does not validate CaMFuzz in inter-host deployments or cover failures whose triggers depend on cross-machine networking, network topology, cluster schedulers, or GPU scales beyond the evaluated server. Such failures may involve communication behaviors and deployment conditions that cannot be reproduced faithfully in our current environment. Although the architecture can accommodate a multi-host runner without changing the core testing methodology, implementing and systematically evaluating such support remains future work. Extending the evaluation to multi-host clusters would also allow us to study whether additional configuration–model interactions emerge only under cross-machine execution. Accordingly, the results in this paper should be interpreted within the single-host setting, and we make no experimentally validated claims regarding the applicability of CaMFuzz to inter-host distributed training.

Limitation of CMSpec. The CMSpec relies on static analysis of framework source code to extract configuration constraints, model profiles, and explicit configuration–model dependencies. While effective for constraints expressed through assertions, exception raising, and conditional branches, it may miss semantics established through complex dynamic mechanisms, such as hardware-dependent checks, factory patterns, or dynamic dispatch. As a result, some valid model-side requirements or configuration–model dependencies may be incomplete. Future work may incorporate lightweight dynamic analysis, such as tracing configuration validation and model initialization during sample runs, to capture semantics that static analysis cannot recover. Additionally, LLM-assisted extraction could help interpret complex code patterns like factory methods and dynamic dispatch, further improving the completeness of the CMSpec.

Supporting More Bug Types. Currently, CaMFuzz primarily focuses on detecting DTBs that manifest as crashes and hangs during training execution. However, distributed training frameworks are also susceptible to other types of defects, such as silent corruption and correctness issues that only become apparent through degraded model accuracy after convergence. Based on our observations, these defects exhibit diverse symptoms and require specialized oracles for detection. We attempted to integrate TrainCheck, an existing tool for detecting silent training errors, into CaMFuzz as an oracle to capture such defects. While TrainCheck reported numerous invariant violations, manual analysis revealed that all of them were false positives. However, this does not mean that CaMFuzz is unable to trigger silent bugs, but rather that appropriate oracles are required to capture them. Our evaluation on historical DTBs demonstrates that CaMFuzz can effectively detect silent errors when equipped with appropriate oracles, successfully reproducing 37 out of 40 silent bugs. Therefore, designing more precise and reliable oracles to extend CaMFuzz's detection capability to these additional defect types remains an important direction for future work.

8 Related Work

Deep Learning Framework Testing. Existing testing tools for deep learning libraries, such as CRADLE [43], LEMON [59], Muffin [14], DocTer [62], NNSmith [33], and FreeFuzz [61], mainly target single-node frameworks by using techniques such as differential testing, metamorphic

testing, constraint mining, and API usage analysis to detect bugs in systems like TensorFlow and PyTorch. More recent work has considered distributed training correctness [6–8, 21, 22, 34, 36, 37, 42, 54, 58, 63, 67, 69, 75]: TrainCheck [20] and TrainVerify [35] check training consistency, D^3 [55] detects DTBs via equivalence between distributed and single-node execution, and TTrace [19] detects and localizes silent bugs by comparing distributed intermediate tensors with a trusted single-device reference. However, these approaches still provide limited coverage of configuration parameters and model diversity, making it difficult to expose bugs that depend on specific configuration–model interactions. In contrast, CaMFuzz systematically explores the joint space of training configurations and model structures through CMSpec guided configuration generation and train code coverage guided model mutation, achieving broader coverage and finding more bugs.

Configuration Testing. Configuration testing aims to detect bugs in system configuration handling logic [4, 16, 23, 56, 57, 64, 65, 68, 70–72]. Tools such as Ctest [50], ConfTest [29], and ECFuzz [26] generate diverse configuration inputs through mutation-based strategies and model the dependencies between configuration parameters. However, these tools are designed for general software systems and do not address the unique interaction between configurations and model structures that characterizes distributed training frameworks. Additionally, their modeling approaches often rely on manual effort, making them both resource-intensive and difficult to adapt to new domains. Unlike existing configuration testing tools, CaMFuzz goes beyond testing configuration logic alone by incorporating configuration–model interaction fuzzing to address the configuration–model co-dependence that is unique to distributed training.

9 Conclusion

In this paper, we present CaMFuzz, a testing framework for detecting distributed training bugs through configuration–model interaction exploration, and experimentally validate it in a single-host, multi-GPU setting. CaMFuzz constructs a CMSpec to generate valid configurations together with compatible model-side requirements, and employs coverage-guided fuzzing to mutate model structures and explore implicit configuration–model interactions. We implemented CaMFuzz and evaluated it on four widely used distributed training frameworks using a single-host, eight-GPU server. Compared with representative baselines, CaMFuzz achieved 75.5%–90.6% higher train logic coverage and detected 20 previously unknown DTBs, all confirmed and fixed by developers. These results validate CaMFuzz 's effectiveness in the single-host, multi-GPU setting, while inter-host networking and large-scale cluster failures remain outside the experimentally validated scope.

Acknowledgments

We sincerely thank the reviewers for their valuable comments on this paper. This research is sponsored by the NSFC Program (No.62525207, U2441238).

References

[1] 523997931. 2025. [BUG]try to train lora but the lora parameter seem not be updated, how to check the reason. https://github.com/deepspeedai/DeepSpeed/issues/7632.

[2] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 472–487.

[3] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[4] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-case prioritization for configuration testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 452–465.

[5] chotzen. 2026. [BUG] Token routing probability all-gather precision in token_dispatcher causes differing results between EP ranks. https://github.com/NVIDIA/Megatron-LM/issues/1421.

[6] Weihao Cui, Ji Zhang, Han Zhao, Chao Liu, Wenhao Zhang, Jian Sha, Quan Chen, Bingsheng He, and Minyi Guo. 2025. Xputimer: Anomaly diagnostics for divergent llm training in gpu clusters of thousand-plus scale. *arXiv preprint arXiv:2502.05413* (2025).

[7] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, et al. 2025. Minder: Faulty machine detection for large-scale distributed model training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 505–521.

[8] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, et al. 2025. Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 254–269.

[9] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, et al. 2024. Efficient training of large language models on distributed infrastructures: a survey. *arXiv preprint arXiv:2407.20018* (2024).

[10] Hugging Face. 2026. Accelerate. https://github.com/huggingface/accelerate.

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[12] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[13] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[14] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 1418–1430.

[15] Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. 2021. Fastmoe: A fast mixture-of-expert training system. *arXiv preprint arXiv:2103.13262* (2021).

[16] Peng Huang, William J Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. Confvalley: A systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[18] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[19] Haitian Jiang, Shaowei Zhu, Zhen Zhang, Zhenyu Song, Xinwei Fu, Zhen Jia, Yida Wang, and Jinyang Li. 2025. TTrace: Lightweight Error Checking and Diagnosis for Distributed Training. *arXiv preprint arXiv:2506.09280* (2025).

[20] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. 2025. Training with confidence: Catching silent errors in deep learning training with automated proactive checks. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 313–329.

[21] Zhihan Jiang, Junjie Huang, Guangba Yu, Zhuangbin Chen, Yichen Li, Renyi Zhong, Cong Feng, Yongqiang Yang, Zengyin Yang, and Michael Lyu. 2025. L4: Diagnosing large-scale llm training failures via automated log analysis. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 51–63.

[22] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[23] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 157–166.

[24] KyleMylonakisProtopia. 2025. [BUG] Deepspeed Crashes when using MoE, Stage 2 Offload with DeepSpeedCPUAdam. https://github.com/deepspeedai/DeepSpeed/issues/5203.

[25] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[26] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. Ecfuzz: Effective configuration fuzzing for large-scale systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[27] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*. 766–775.

[28] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[29] Wang Li, Shanshan Li, Xiangke Liao, Xiangyang Xu, Shulin Zhou, and Zhouyang Jia. 2017. Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. 88–97.

[30] Feng Liang, Zhen Zhang, Haifeng Lu, Victor Leung, Yanyi Guo, and Xiping Hu. 2024. Communication-efficient large-scale distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2404.06114* (2024).

[31] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. 2024. {nnScaler}:{Constraint-Guided} Parallelization Plan Generation for Deep Learning Training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 347–363.

[32] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. 2024. Aceso: Efficient parallel DNN training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 163–181.

[33] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[34] Xuanzhe Liu, Diandian Gu, Zhenpeng Chen, Jinfeng Wen, Zili Zhang, Yun Ma, Haoyu Wang, and Xin Jin. 2023. Rise of distributed deep learning training in the big model era: From a software engineering perspective. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–26.

[35] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. 2025. Trainverify: Equivalence-based verification for distributed llm training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 237–253.

[36] Jeffrey Jian Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. 2025. Understanding silent data corruption in LLM training. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 20372–20394.

[37] Xiaoxue Ma, Wanwei Zhan, Jiale Chen, Yishu Li, Jacky Keung, and Federica Sarro. 2025. A Comprehensive Study of Bugs in Modern Distributed Deep Learning Systems. *arXiv preprint arXiv:2512.20345* (2025).

[38] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[39] Microsoft. 2026. DeepSpeed. https://github.com/deepspeedai/DeepSpeed.

[40] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[41] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–15.

[42] Deep Pankajbhai Mehta. 2026. Placement Semantics for Distributed Deep Learning: A Systematic Framework for Analyzing Parallelism Strategies. *arXiv e-prints* (2026), arXiv–2601.

[43] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038.

[44] qby10. 2024. [BUG] pipeline_paralle is not available when pp_size > 2. https://github.com/NVIDIA/Megatron-LM/issues/873.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*. IEEE, 1–16.

[46] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international*

conference for high performance computing, networking, storage and analysis. 1–14.

[47] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[48] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[50] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 735–751.

[51] tangefly. 2025. [BUG]MoE router parameters are forced to bf16 under DeepSpeed bf16, causing dtype mismatch in fp32 routing logic. https://github.com/deepspeedai/DeepSpeed/issues/7747.

[52] thuwzt. 2024. [BUG] MoE load balancing loss is accumulated twice when using activation checkpointing. https://github.com/NVIDIA/Megatron-LM/issues/1330.

[53] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.

[54] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, et al. 2025. Robust llm training infrastructure at bytedance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 186–203.

[55] Jiannan Wang, Hung Viet Pham, Qi Li, Lin Tan, Yu Guo, Adnan Aziz, and Erik Meijer. 2024. D 3: Differential testing of distributed deep learning with model generation. *IEEE Transactions on Software Engineering* 51, 1 (2024), 38–52.

[56] Shuai Wang, Xinyu Lian, Qingyu Li, Darko Marinov, and Tianyin Xu. 2024. Ctest4j: A practical configuration testing framework for java. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 562–566.

[57] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. 2023. Understanding and detecting on-the-fly configuration bugs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 628–639.

[58] Zhanghan Wang, Ding Ding, Hang Zhu, Haibin Lin, and Aurojit Panda. 2025. Verify Distributed Deep Learning Model Implementation Refinement with Iterative Relation Inference. *arXiv preprint arXiv:2508.09505* (2025).

[59] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 788–799.

[60] wangyan mms. 2025. [BUG] Integer overflow in FusedAdam silently prevents weight updates for large tensors (>= 2^31 elements). https://github.com/deepspeedai/DeepSpeed/issues/7640.

[61] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[62] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 176–188.

[63] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. 2024. {SuperBench}: Improving cloud {AI} infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 835–850.

[64] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 619–634.

[65] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 244–259.

[66] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663* (2021).

[67] Zhiyi Yao, Pengbo Hu, Congcong Miao, Xuya Jia, Zuning Liang, Yuedong Xu, Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, et al. 2025. Holmes: Localizing irregularities in {LLM} training with mega-scale {GPU} clusters. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 523–540.

[68] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 159–172.

[69] Xiao Yu, Haoxuan Chen, Feifei Niu, Xing Hu, Jacky Wai Keung, and Xin Xia. 2025. Towards Understanding Bugs in Distributed Training and Inference Frameworks for Large Language Models. *arXiv preprint arXiv:2506.10426* (2025).

[70] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. 28–28.

[71] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 687–700.

[72] Sai Zhang and Michael D Ernst. 2013. Automated diagnosis of software configuration errors. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 312–321.

[73] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[74] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[75] Kahfi S Zulkifli, Wenbo Qian, Shaowei Zhu, Yuan Zhou, Zhen Zhang, and Chang Lou. 2025. Verifying Computational Graphs in Production-Grade Distributed Machine Learning Frameworks. *arXiv preprint arXiv:2509.10694* (2025).