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 con gurations, and employs con guration model fuzzing to e ectively 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 con rmed and xed by developers. We open-source CaMFuzz[1] for broader practical use.

2 Background

Figure 1. Overview of how distributed training frameworks coordinate training con gurations 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 con guration and a model de nition, and automatically manage the full distributed execution lifecycle. During initialization, the framework parses the con guration, sets up the distributed environment, restructures the model according to the speci ed 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 con gurations 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, a ecting communication patterns, memory management, optimizer behavior, and synchronization mechanisms [31, 32, 41, 47, 69, 73, 74]. Complementing this, model structures de ne the neural network architecture to be trained, encompassing layer types, layer depth, hidden dimensions, and connectivity patterns. The model de nition 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 con guration and model structure jointly determine which execution paths are activated, neither can be considered in isolation [2, 11, 15, 18, 40, 53, 66]. A con guration that behaves correctly with one architecture may trigger a latent bug when paired with a di erent model structure. Errors that never appear under a single con guration or a xed model often manifest only when particular con gurations and model characteristics co-occur, leading to crashes, silent numerical corruption, or training hangs. Because these conditions require speci c combinations of con guration parameters, model properties, and runtime interactions, existing testing techniques rarely explore this execution space, leaving con guration model interaction bugs largely untested.

Throughout this paper, an explicit con guration model dependency is a model requirement imposed by a training con guration and required for the con guration model pair to successfully initialize and execute. An implicit con guration model interaction refers to a valid con guration model pair whose combined properties jointly activate a speci c runtime execution path or behavior that would not be determined or exercised by either the training con guration 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 o cial 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 con rm the defect through source-code analysis supported by the discussion or patch. After ltering, we identi ed 133 con rmed DTB cases as shown in Table 1.

Table 1. Number of DTBs identi ed 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 #7547 [53] 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 over ow 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 classi cation 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 con gurations satisfying explicit con guration 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 con gurations; otherwise, training fails at initialization and the bug-triggering execution path is never reached. Validity here means that a con guration must satisfy both intra-con guration constraints (e.g., enabling CPU o oading requires compatible optimizer types) and explicit con guration 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 con gurations span multiple dimensions of distributed training, including parallelism strategies (DP, TP, PP, EP), memory optimizations (ZeRO stages, CPU o oad, activation checkpointing), numerical precision (fp16, bf16, fp8), communication optimizations (gradient overlap, contiguous gradients), and checkpointing mechanisms (async checkpointing, distributed checkpoints). For instance, DeepSpeed bug #5204 [24] is triggered only when ZeRO-2 CPU o oading is enabled together with an MoE model under `ep_size > 1`, where the con guration's explicit dependency on MoE layers must be satis ed for the faulty expert gradient synchronization path to be reached. The case shows that DTBs are frequently hidden under valid con gurations whose explicit dependencies on model structures are satis ed, and constraint-unaware con guration testing that ignores such dependencies is insu cient to expose them.

Table 3. Training con guration dependency of DTBs.

Dependency	DeepSpeed	Accelerate	ColossalAI	Megatron	Total (%)
Requires speci c con g	57	27	5	35	124 (93%)
No speci c con g needed	2	2	3	2	9 (7%)

Finding 3: Implicit con guration 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 con guration's explicit dependencies, while Explicit + Implicit DTBs additionally require speci c model structures to activate the faulty execution path through implicit dependencies. Nearly half of DTBs (45%) fall into the latter category, where the same con guration activates di erent execution paths depending on the model's structural properties, exposing di erent bugs. Even when explicit dependencies are satis ed and training proceeds normally, speci c model characteristics are still required to enter the faulty code paths. For example, Megatron-LM bug #1421 [51] is triggered only when MoE layers activate the token dispatcher path, and DeepSpeed bug #7632 [61] manifests only when LoRA layers invoke the framework's special handling for trainable parameters. These results indicate that triggering DTBs requires not only valid con gurations but also speci c model structures that activate the corresponding execution paths.

Table 4. Triggering conditions of DTBs categorized by con guration 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 con guration model interactions can lead to severe consequences in distributed training frameworks. One such example is a DTB in DeepSpeed (Issue #5240 [23]), where the interaction between ZeRO Stage-2 CPU o oading 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 x.

When CPU o oading 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 o oading and DeepSpeedCPUAdam. Forward and backward passes complete normally, but expert gradient norms are computed and stored on CPU due to o oading. 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 x (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 o oading, nor when enabling CPU o oading 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 o oading 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 ndings: 1) DTBs are reachable only when training proceeds with valid con gurations satisfying explicit con guration model dependencies. Here, ZeRO-2 CPU o oading 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 con gurations with compatible model-side requirements. 2) Even when explicit dependencies are satis ed, different model structures activate di erent 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 re ected in the con guration. This motivates CaMFuzz's con guration 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 con gurations and model structures. Two mainstream categories of methods dominate current practice, yet both fail to systematically explore the con guration model interaction space where many DTBs arise.

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

under the same con guration, and bugs like the motivating example require speci c model characteristics that xed models do not provide. Consequently, the 45% of DTBs that depend on implicit con guration 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 di erential 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 con gurations such as parallelism strategies, ZeRO stages, or mixed-precision settings. As a result, the generated models are executed under default or trivial con gurations that fail to activate the distributed training code paths where DTBs reside. According to Findings 2 and 3, neither valid con gurations satisfying explicit dependencies nor the implicit con guration model interactions that trigger 45% of DTBs can be exercised by model generation alone, leaving the con guration dimension of the interaction space entirely unexplored.

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

CaMFuzz addresses this challenge by tightly coupling CMSpec-guided con guration generation with dependency-preserving, coverage-guided model exploration. CMSpec extracts con guration constraints, model pro les, and explicit con guration model dependencies, and generates each training con guration together with its model-side requirements. For each generated con guration, 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 con guration model interactions without sacri cing 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 con guration or model exploration is likely to miss.

4 CaMFuzz Design

4.1 CaMFuzz Work ow

Figure 4 illustrates the work ow of CaMFuzz, which consists of two main phases. The rst phase focuses on Con guration Model Speci cation Construction: (1) The CMSpec Constructor analyzes the target framework's source code. (2) It extracts training con guration constraints and model pro les from the code. (3) These are used to build the CMSpec. (4) The CMSpec generates valid training con gurations that satisfy the extracted constraints.

Figure 4. Work ow of CaMFuzz. It includes two main phases: (1) Con guration Model Speci cation Construction and (2) Con guration Model Interaction Fuzzing.

Based on the generated con guration, CaMFuzz proceeds to the second phase, Con guration Model Interaction Fuzzing, consisting of the following steps: (5) A compatible model template is selected based on the structural requirements imposed by. (6) The Con guration Model Fuzzer synthesizes a concrete model instance from the selected template. (7) The generated con guration 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 Identi er traces the execution of the framework's core distributed training modules. (9) The trace is ltered 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).