RECOVER: Resilient LLM Pre-Training System via Fault-Tolerant Collective and Versatile Workload

Ziyue Liu[1], Zhengyang Wang[1], Ruijie Zhang[1], Avinash Maurya[2], Hui Zhou[2], Paul Hovland[2], Sheng Di[2], Franck Cappello[2], Bogdan Nicolae[2], Zheng Zhang[1]
[1]University of California at Santa Barbara; [2]Argonne National Laboratory
ziyueliu@ucsb.edu, zhengzhang@ece.ucsb.edu

Abstract

Pre-training large language models on massive GPU clusters has made hardware faults routine rather than rare, driving the need for resilient training systems. Yet existing frameworks either focus on specific parallelism schemes or risk drifting away from a failure-free training trajectory. We propose RECOVER, a resilient LLM pre-training system that upholds a single invariant: each iteration keeps the number of microbatches constant, ensuring per-iteration gradients remain stochastically equivalent to a failure-free run. The framework is organized as three decoupled protocol layers: (1) Fault-tolerant collectives that isolate faults from propagating across replicas; (2) in-step fine-grained recovery that preserves intra-iteration progress and prevents gradient corruption; (3) versatile-workload policy that dynamically redistributes microbatch quotas across the survivors. The design is parallelism-agnostic, integrating directly with both 3D parallelism and Hybrid Sharded Data Parallel (HSDP) as a drop-in substrate. We evaluate our implementation on end-to-end pre-training tasks for up to 512 GPUs, RECOVER successfully preserves the training trajectory from a failure-free reference despite 256 GPUs lost spread across the run. For comparison with checkpoint-and-restart baselines, RECOVER demonstrates $2.23\times$ higher effective throughput after successive failures. This advantage results in RECOVER processing 74.9% more tokens at 234 GPU-hours, with the gap widening as the training prolongs [1].

1 Introduction

The increasing demand of pre-training frontier large language models (LLMs) has driven industrial high-performance computing (HPC) platforms to rapidly scale to over $O(100\text{k})$ GPUs. For instance, LLaMA-4 was trained on 100k GPUs [9]; Grok-4 was trained on 200k GPUs [41], while xAI was reported to build a cluster with 1 million H100 GPUs [7]. At such scales, the mean-time-between-failures (MTBF) shrinks inversely with the GPU count: from 3 hours at 16k GPUs [8] to only 18 min at 100k GPUs [30], and projected below 5 min at 600k [18]. Thus, traditional checkpoint-restart techniques are bottlenecked by *recovery overhead* (re-initialize the communication backend and training pipeline, load model parameters and optimizer state, replay from last checkpoint), which is reported to be 10 min on production 100k-GPU clusters where failure lands every 18 min [30]. That is, system makes progress in only 8 of every 18 min, $> 50\%$ of total GPU hours are wasted. Despite significant progress in reducing checkpoint overhead [23, 5, 39, 22, 26, 36, 37, 21], we still approach zero useful throughput when MTBF approaches the restart overhead, motivating the need for alternatives that keep the job alive across failures, which we call *forward recovery* methods.

Limitations of state of the art. Although forward recovery has been studied in the AI and HPC community, existing efforts fall short along three dimensions. First, prior work is often *layer-isolated*:

[1]Code is available at https://github.com/alvin-zyl/ReCoVer.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

resilience of the communication layer [20, 2, 1, 3, 17, 19] is well studied but insufficient in today's highly structured LLM pre-training stacks [32, 25, 11, 29, 44, 14] where communication patterns involve more complex multi-collective, multi-process-group with strict performance guarantees. Second, prior work lacks *versatility*: pipeline-centric systems [34, 12, 6] embed shadow stages into the pipeline schedule and are tightly coupled to that single parallelism flavor, failing to extend to other parallelism schemes, such as HSDP [44]. Third, in a quest to lower performance overheads, some approaches do not preserve *computational equivalence*: [30] continues training with reduced number of microbatches after a failure, which shifts the gradient-noise scale and induces loss spikes that, under frequent failures, may accumulate and drift away from the failure-free baseline. A recent proposal [18] pursues forward recovery via data shard replication, but its performance is only validated via simulation without providing a real implementation, and the method undergoes a large factor of redundant computation once a failure occurs.

Contributions. We present RECOVER, a resilient LLM pre-training system that closes the three gaps above. RECOVER delivers *forward recovery* that (i) is *extended* across the full pre-training stack rather than the communication layer alone, (ii) is *versatile* across different parallelism schemes, and (iii) preserves *computational equivalence* to the failure-free trajectory, regardless of when or where a failure occurs. This is realized as a three-layer fault-tolerant protocol that, under any failure schedule, upholds a single invariant: every iteration commits gradients from the same number of microbatches as its failure-free reference, keeping each optimizer update stochastically equivalent, without rolling back, replaying, or paying for idle replication. We summarize our contributions as follows:

- **Analysis of fault-tolerant pre-training (Section 3).** We examine a synchronous iteration under device loss, analyze how a failure propagates through collective communication, and identify five challenges that any forward-recovery system must address.
- **Bottom layer: contains failures locally (Section 4.1).** Going beyond communication-layer-only resilience, RECOVER extends fail-continue guarantees to the multi-collective, multi-process-group structure of modern LLM pre-training: it repairs communicators in-place over survivors and constructs a collectively agreed post-failure consistent view.
- **Middle layer: enables *forward recovery* (Section 4.2).** RECOVER orchestrates training-level recovery within the failed iteration, restoring affected gradients to their pre-reduction state to prevent corruption and preserve survivor's intra-iteration progress, without rolling back or replaying.
- **Top layer: guarantees *computational equivalence* (Sections 4.3 and F).** RECOVER dynamically redistributes microbatch quotas across survivors to keep the total number of microbatches constant, thus being stochastically equivalent to its failure-free trajectory without pre-allocated idle replicas.
- **Integration and evaluation across parallelism stacks (Sections 4.4 and 5).** Against checkpoint-restart baselines, RECOVER delivers $2.23\times$ higher effective throughput under successive failures and processes 74.9% more tokens at 234 GPU-hours, with the gap widening as training prolongs, while preserving training trajectory to its failure-free reference despite losing up to 256 GPUs.

2 Related Work

Failure issues in LLM pre-training systems. Modern LLM pre-training rests on a decade of parallel-training infrastructure [31, 32, 25, 11, 24, 29, 44, 14, 4, 40]. All devices rendezvous at gradient synchronization, so a single device's failure terminates the job. Production-cluster studies [13, 10, 16, 42, 30] report mean-time-between-failures (MTBF) to be tens of minutes at the 100k-GPU scale. At such scale, effective training time is less than even 50% [30], making fault tolerance a first-class system requirement. This issue is expected to become much more significant as the cluster continues to scale up and becomes more heterogeneous.

Checkpoint and restart. The prevailing resilience strategy is periodic checkpointing of model, optimizer, and dataloader state combined with job restart from the latest checkpoint on failure [25, 14]. Subsequent systems lower checkpoint overhead through frequent fine-grained snapshots [23], in-memory and hierarchical replication [39, 5], lazy asynchronous persistence [22, 26, 21], and unified multi-framework checkpoint management [36]. All retain the restart-and-replay semantics.

Fault-tolerant frameworks. A second line of research keeps the job alive across failures. At the communication layer, User-Level Failure Mitigation (ULFM) [20, 2, 1, 3, 17] lifts MPI from fail-stop to fail-continue; Early work [19] integrates it with Horovod [31] for DP-only deep learning applications.

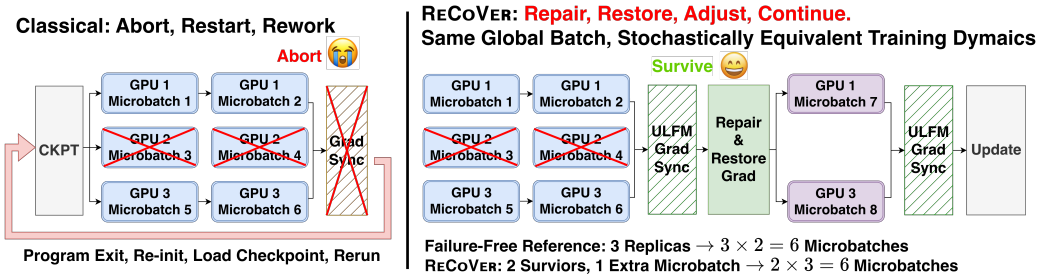


Figure 1: Comparison of a classical synchronous iteration (left) and RECOVER (right) under a mid-iteration failure. The classical approach aborts all replicas; RECOVER localizes and repairs broken gradient sync in place, adjusts survivor workload, and completes with the same microbatch count, preserving stochastically equivalent update.

R^{2}CCL [38] handles NCCL link failures, not process crashes. Pipeline-centric systems [34, 12, 6] make pipeline stages the unit of redundancy, running shadow stages or rebuilding the schedule around survivors. In production, FTAR [30] extends NCCL with a revoke-and-rejoin path and reports HSDP pre-training through sustained failures by re-provisioning a replica and asynchronously rejoining it.

3 Fault-Tolerant Challenges in LLM Pre-Training

Term definitions. To avoid confusion, we define several terms used throughout this paper. **Replica:** one complete copy of the model. **Rank:** the index of a device among a certain group. **Microbatch:** one forward-backward pass done by a replica. **Gradient synchronization:** gradient all-reduce across the replicas. **Gradient accumulation:** gradient aggregation of G microbatches on each replica before gradient synchronization. **Iteration:** the full cycle of gradient accumulation, synchronization, model update.

Iteration anatomy and what happens after failure. During an iteration, replicas compute microbatches independently and remain unaware of remote failures until gradient synchronization, when failures are detected. As shown in Figure. 1, traditional checkpoint-restart approaches must then abort all replicas and discard all progress since the last checkpoint, despite failures typically affecting only a small number of replicas (often just one). We therefore argue that resilience at hyperscale pre-training needs to protect this synchronization point, such as to be able to contain failures locally to the affected replicas, which is a premise of forward recovery.

Five challenges. *C1: standard communication backends do not survive rank loss.* Nvidia Collective Communication Library (NCCL) is the de facto choice for deep learning applications and classical Message Passing Interface (MPI) is the mainstream communication protocol in high-performance-computing (HPC) community, yet they both have no application-facing way to complete an in-flight collective under failure or form a new process group over survivors. Recent works [30, 38, 20, 2] have made prominent progress, however, none of which have become standard in LLM applications. A clean post-failure communicator (C1) is necessary but not sufficient. *C2: drop-and-go is not computationally equivalent.* After satisfying C1, one solution would be to drop the faulty replica and continue with the survivors. However, such approach would use fewer microbatches in the gradient synchronization, shifting the gradient-noise scale and inducing loss spikes that deviate from the baseline configuration. *C3: asynchronous rejoin does not solve computational equivalence.* Even if replicas may rejoin later [30], under frequent failures, a degraded mode with smaller world size would be the norm rather than exception, thus suffering from the same issues as drop-and-go. *C4: hot spares are rigid and wasteful.* Using spare replicas addresses C2–C3 at the cost of resource redundancy, which is expensive for large models that need many GPUs for each replica. *C5: parallelism-specific designs lack versatility.* A versatile protocol must decouple the cross-replica failure-recovery logic from the intra-replica communication structure to allow seamless forward recovery integration.

4 The RECOVER Framework

Design overview. RECOVER is a three-layer fault-tolerant protocol (overview in Figure 1, details in Algorithm 1). At the bottom layer, a fault-tolerant collective ULFM_ALLREDUCE (Section 4.1) verifies

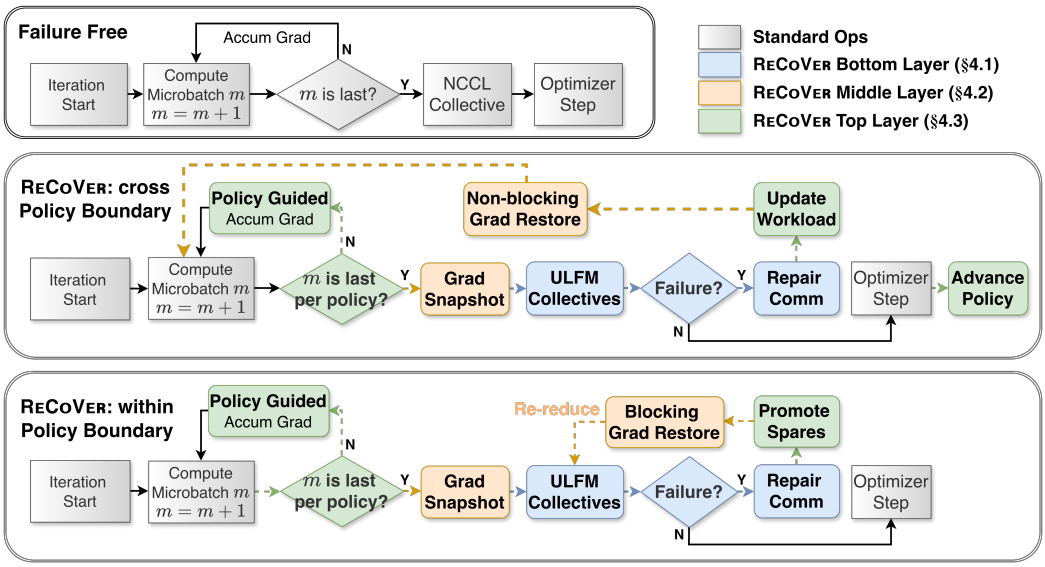


Figure 2: Flowchart of a ReCoVer iteration and how its three-layer protocol interacts.

communicator health *before* each all-reduce. Upon failure, it repairs the communicator over surviving ranks and either early-returns or performs a guarded reduction, ensuring no fatal errors and returning globally consistent status for all callers. The middle layer, *in-step fine-grained recovery* (Section 4.2), uses this signal to restore gradients to their pre-reduction state, preventing inconsistent reductions and preserve survivor's intra-iteration progress. The top layer, *versatile workload* (Section 4.3), a policy that dynamically adjusts replica workloads after failures to maintain a constant per-step microbatch count.

Claim. Together, the three layers ensure that, as long as one replica survives, each ReCoVer iteration is stochastically equivalent to a failure-free run: every iteration aggregates the same number of microbatch gradients before the optimizer step. Failed replicas' data partitions are dropped, while survivors process their partitions faster. Given the large pre-training data stream, this is equivalent to a different random shuffle of the same corpus, so per-iteration gradients follow the same distribution as in the failure-free case. Section F formalizes this.

Algorithm 1 A ReCoVer iteration.

Require: policy P; replica role ρ; policy-assigned per-role workload $P(\rho)$; target total workload B; bucket bookkeeping $\mathcal{B}$; communicator epoch ϵ_{cur}; intra-replica group PG_{intra}; cross-replica group PG_{cross}

$\mathcal{B} \leftarrow \varnothing$; zero gradients; $m \leftarrow 0$.
while microbatch index $m < P(\text{major})$ **do**
  $m \leftarrow m + 1$; run forward and backward.
  if $m \leq P(\rho)$ **then** locally accumulate gradient;
  else zero this microbatch's gradient.
  end if
  if $m = P(\text{major})$ **then**
    for all newly produced gradient bucket b **do**
      snapshot $S(b) \leftarrow b$; tag $\epsilon(b) \leftarrow \epsilon_{\text{cur}}$;
      append $(b, S(b), \epsilon(b))$ to $\mathcal{B}$.
      if $\rho \in$ {major-spare, minor-spare} **then**
        zero b.
      end if
      post ULFM_ALLREDUCE on PG_{cross}.
    end for
    post NCCL barrier on PG_{intra}.
    post ULFM_CONSENSUS on PG_{cross}.
    if any ULFM collective returns failure **then**
      call HANDLE_WORK_FAILURE.
      call GRADIENT_RESTORATION.
      $P, \rho \leftarrow$ POLICY_ADJUSTMENT.
    end if
  end if
end while
divide accumulated gradient by B; optimizer step.
if crossed a policy boundary **then**
  $P, \rho \leftarrow$ POLICY_ADVANCEMENT.
end if

4.1 Bottom Layer: ULFM-Guarded Fault-Tolerant Collectives

Goal and primitives. The bottom layer's goal is to provide resilient collective primitives for the rest of the framework to survive rank loss: every caller eventually returns, either completing the reduction on a repaired communicator or surfacing a collectively agreed consistent failure signal. ReCoVer adds two such primitives to PyTorch's backend (Figure 3): ULFM_ALLREDUCE, a fault-tolerant all-reduce used wherever the application would issue a standard cross-replica one; and ULFM_CONSENSUS, a barrier-like collective that guarantees the consistency of

replicas (e.g., all ranks of a replica either succeed together or discover failures together) and ensures all replicas agree on the same view for upper-layer's recovery logic (Sections 4.2 and 4.3).

Why ULFM is a good foundation. User-Level Failure Mitigation (ULFM) [20] extends MPI with the minimal semantics these primitives need: no MPI call blocks indefinitely after a failure; it either succeeds or returns a typed error. Unlike NCCL or conventional MPI, where any rank loss aborts the job, ULFM exposes failures at the communicator level and leaves recovery path to the application. We build on two of its collective routines: (1) `MPIX_Comm_agree`, which implements fault-tolerant consensus (i.e., returns success only if all communicator members are alive); and (2) `MPIX_Comm_shrink`, which builds a new communicator that includes only surviving members. ULFM is intentionally minimal, focusing solely on collective membership management. Any buffers and data involved in failed collectives are left in an unspecified state. RECOVER guarantees recovery of these data structures at upper layers, which is only feasible based on the strong foundation of ULFM that keeps the application alive.

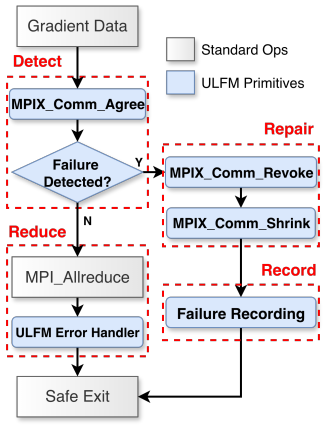


Figure 3: RECOVER bottom layer.

Building the primitives. ULFM_ALLREDUCE (Figure 3) composes these routines under a deliberately conservative contract: *avoid issuing an all-reduce on a communicator on which some processes might have already died*. Given a tensor, a reduction operator, and a process group PG, it runs four steps: (1) *Detect.* Call `MPIX_Comm_agree` on PG; if successful, jump to step 4; (2) *Repair.* A failure happened; revoke PG's communicator and call `MPIX_Comm_shrink` to form a communicator over survivors; (3) *Record.* On the repaired communicator, agree on application-specific failure knowledge and return early; (4) *Reduce.* Issue the MPI all-reduce on the validated communicator. Similarly, ULFM_CONSENSUS follows steps 1–3 for a barrier-like failure detection and recovery primitive.

Why these primitives address *C1*. Failures are safely isolated to faulty replicas (right half of Figure 1), and the upper layers choose how to proceed instead of being forced to abort. This also lays the groundwork for *C5*: communication resilience is exposed as a drop-in collective interface, independent from any specific intra-replica communication structure, so any layer above can adopt it without changing its parallelism. The remaining challenges *C2–C4* and the full decoupling needed for *C5* are addressed by the upper layers under LLM pre-training settings.

4.2 Middle Layer: In-Step Fine-Grained Failure Recovery

Why fine-grained recovery matters at scale. A single LLM pre-training iteration spans many microbatches, yielding global batches of hundreds to thousands of millions of tokens that scale with total training tokens [43]. This regime is not only viable but desirable, as it amortizes latency-bound cross-replica communication and improves GPU utilization. Consequently, one iteration may last minutes, while failures occur every tens of minutes on a 100K-GPU cluster, making it essential to preserve progress within each iteration.

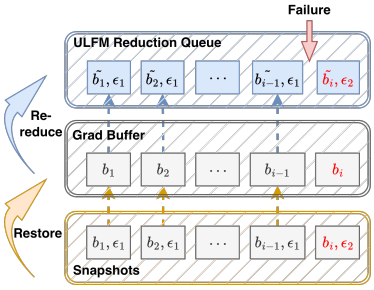


Figure 4: RECOVER middle layer

Failure anatomy inside an iteration. Failures can occur in three cases: **(a) Before gradient synchronization.** Buckets contain only local gradients; the first ULFM_ALLREDUCE uses a shrunk communicator, so no reductions span different memberships. **(b) After synchronization, before the optimizer step.** All reductions complete in the original world, so gradients remain valid and the failure can be handled in the next iteration safely. **(c) During synchronization.** Some gradients were reduced in the original world and include contributions from replicas absent in the shrunk world. Above-iteration schemes like FTAR [30] must discard progress, whereas RECOVER avoids this limitation via ULFM's communicator-level fault tolerance and a per gradient-bucket bookkeeping.

Per gradient-bucket bookkeeping. Before each gradient all-reduce, RECOVER snapshots the bucket's pre-reduce state along with a monotonic *world epoch* that increments on each communicator shrink. Upon failure, only snapshots from smaller *world epoch* are restored. This fine-grained

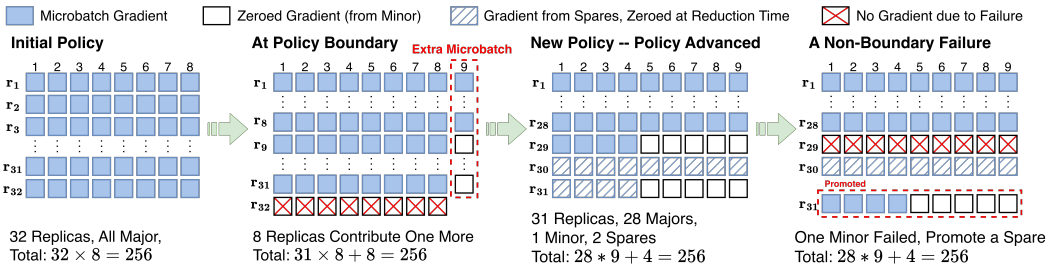


Figure 5: Versatile workload across two failures. (i) Pre-failure: all replicas are major. (ii) First failure crosses policy boundary: no spares available, compute extra microbatches. (iii) Policy advanced: 28 majors, 1 minor, and 2 spares. (iv) Second failure within policy boundary: spares available, promote.

tracking is enabled by ULFM's communicator-level failure detection and recovery, preserving the process group context—unlike [30], which requires tearing down and rebuilding the group.

Blocking and non-blocking restoration. RECOVER selects between two restoration strategies based on whether the iteration proceeds to the optimizer step or defers to a policy boundary (Section 4.3). (1) *Blocking restoration* synchronously restores corrupted buckets and re-reduces them under the shrunk world before the optimizer step. (2) *Non-blocking restoration* schedules the snapshot rewind on a separate CUDA stream and overlaps it with the forward pass of the first extra microbatch at the policy boundary step. This step issues a fresh cascade of all-reduces, eliminating the need to manually re-reduce corrupted buckets. The next subsection's policy layer decides between the two.

4.3 Top Layer: Versatile-Workload for Training Trajectory Preservation

Workload redistribution and justification. The versatile workload relies on two properties of gradient accumulation. *(i) Sum-level fungibility:* an iteration's gradient is the sum of B microbatch gradients, independent of which replica computes each term, any partition of B microbatches across survivors is equivalent. *(ii) Stream-level exchangeability:* when a replica fails, replacement microbatches are drawn from the survivors' own partitions of an effectively infinite, exchangeable data stream, yielding a B-batch that is distributionally equivalent to the original (Section F). Property (i) guides redistribution; property (ii) ensures it preserves the training trajectory. RECOVER leverages both by assigning each survivor replica one of four roles per iteration.

Four replica roles and invariant. Major: contributes G_{cur} microbatches. **Minor:** contributes $R_{\text{cur}} < G_{\text{cur}}$ microbatches to absorb any remainder when $W_{\text{cur}} \times G_{\text{cur}} > B$, with W_{cur} being the current replica count. **Major-/minor-spare:** executes the same workload as its counterpart but zeros its gradient buffer at all-reduce; it does not impact the global batch until promoted after a failure. Let $C_r(t)$ be the number of microbatches replica r contributes at iteration t (zero for spares). RECOVER targets a single invariant across all iterations:

$$\sum_{r \in \text{survivors}(t)} C_r(t) \;=\; W_{\text{init}} \cdot G_{\text{init}} \;=\; B. \tag{1}$$

We call the family of replica-role assignments that maintains Equation (1) the *versatile workload*.

Versatile-workload policy. At each iteration, replica roles adapt to failures based on whether a **Policy Boundary** is reached, i.e., whether available spares can replace failed replicas. If so, spares are promoted to majors/minors and the current accumulation G_{cur} is unchanged. Otherwise, Eq. (1) no longer holds unless a **Policy Boundary Step** being performed. Specifically, for a survivor count W_{cur}, the collectively agreed total contribution from survivors $C_{\text{cur}} = \sum_r C_r(t)$ (obtained from phase 3 in ULFM_ALLREDUCE), and the extra grad-accum steps $G_{\text{ext}} = 1$, the policy increments G_{ext} until $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} \geq B$. In case of an inequality, assign $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} - B$ replicas as **boundary minors**, a temporary role that each of them contributes $G_{\text{ext}} - 1$ extra microbatches. After committing the curren iteration, the policy increments G_{cur} until $W_{\text{cur}} G_{\text{cur}} \geq B$, then sets $N = \lfloor B / G_{\text{cur}} \rfloor$. If $N \cdot G_{\text{cur}} = B$, assign N majors and the rest as major-spares; otherwise assign N majors, one minor with $R_{\text{cur}} = B - N \cdot G_{\text{cur}}$, and distribute remaining replicas as spares.

Illustrative example. Figure 5 instantiates the policy for $W_{\text{init}} = 32$ and $G_{\text{init}} = 8$, so $B = 256$. The four panels trace the policy through one boundary step and a subsequent non-boundary failure.

Pre-failure. All 32 replicas are majors at $G_{\text{init}} = 8$, and $32 \times 8 = 256 = B$.

A replica fails during iteration t. Now $W_{\text{cur}} = 31$ and $C_{\text{cur}} = 31 \cdot 8 = 248$, no spares so it's a policy boundary step: eight survivors contribute one extra microbatch giving $248 + 8 = 256 = B$.

Policy advancement. Incrementing G_{cur} by 1 yields $31 \times 9 = 279 > 256 = B$, so $G_{\text{cur}} = 9$. We have $N = \lfloor 256/9 \rfloor = 28$ and $28 \cdot 9 = 252 \neq 256$, so the assignment is 28 majors at 9 microbatches and one minor at $R_{\text{cur}} = 256 - 252 = 4$ microbatches, with the one of remaining $31 - 29 = 2$ survivors as major spare, one as minor spare.

Next failure, spare available. Each spare runs same microbatches as its non-spare counterparts but has its gradients zeroed at all-reduce time. When the minor r_{29} fails mid-window, the minor spare r_{31} is promoted into the vacated role simply by restoring its gradients (Section 4.2).

4.4 Integration with LLM Pre-Training Frameworks

A realistic LLM pre-training system distributes one replica across many devices, each holding a distinct shard of parameters and gradients. RECOVER naturally lifts onto this setting: every intra-replica rank fires the cross-replica all-reduce and runs the protocol in lockstep. Therefore, RECOVER is agnostic to replica internals and *versatile* across parallelism schemes. However, the inner structure introduces nuances, which we address for **3D parallelism (Data, Tensor, Pipeline)** and **Hybrid Sharded Data Parallel (HSDP)** below.

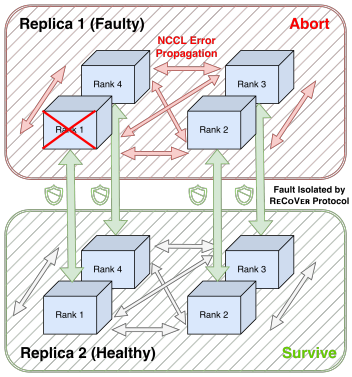


Figure 6: RECOVER integration with model parallelism

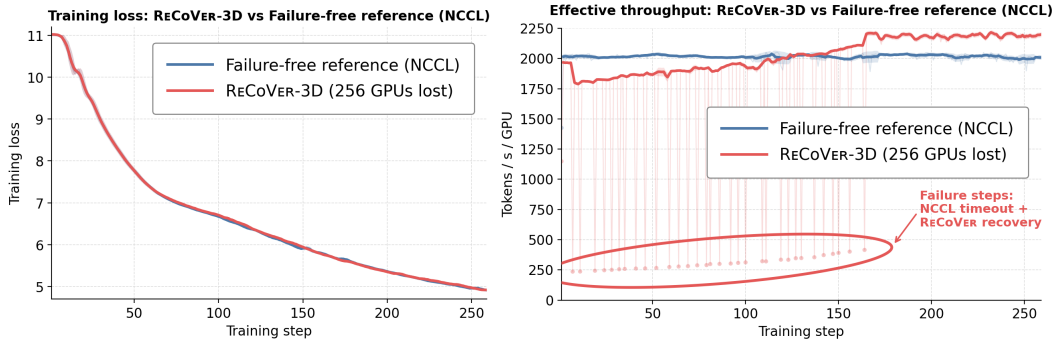
Where the two parallelism substrates agree. In both 3D parallelism and HSDP, losing any rank equals to losing an essential shard of the model, thus invalidating the entire replica, making the replica the atomic survival unit under RECOVER. We keep NCCL for intra-replica communication, while deploying the collectives of Section 4.1 for cross-replica communication. We post an additional replica-consistency gate via a NCCL barrier so that any rank failure will eventually trigger NCCL watchdog timeouts on all intra-replica peers and aborting the replica as a unit. Surviving replicas remain unaffected until the failure detected by their ULFM_ALLREDUCE or ULFM_CONSENSUS, which then repairs and shrinks the communicator to surviving replicas. The remainder of the iteration is then driven by the recovery and policy mechanisms of Sections 4.2 and 4.3. Figure 6 illustrates this asymmetry between intra-replica abortion and cross-replica survival.

Where the two substrates differ. The integrations differ in three minor aspects. *(i) ULFM-guarded cross-replica group.* RECOVER-3D guards the data-parallel all-reduce, whereas RECOVER-HSDP protects the cross-*replicate* group all-reduce. *(ii) Snapshots.* RECOVER-3D snapshots bucketed gradients of its local shard, while RECOVER-HSDP snapshots sharded gradients of flattened FSDP parameters. *(iii) Replica-consistency gate.* RECOVER-3D places the NCCL barrier on the union of TP and PP groups, whereas RECOVER-HSDP barriers on the FSDP shard group.

5 Evaluation

We evaluate RECOVER on the 3D-parallel integration (RECOVER-3D) of Section 4.4, addressing three questions: (i) does it preserve the failure-free optimization trajectory under frequent failures? (ii) what resource overhead does this preservation incur over an ideal never-failing scenario? and (iii) how does it compare to a standard checkpoint-restart [33, 27] approach under single and consecutive failures? We argue that comparing with the failure-free reference is not just enough, but the most challenging setting: a resilient system can do at most as good as its never-failing counterpart. And we only compare with standard checkpoint because they are complementary to RECOVER: an advanced checkpointing method can be combined with our *forward recovery* system to further boost resilience. These settings are also used in the most recent production effort [30], which RECOVER is conceptually complementary with and can be engineered to be combined together. The HSDP integration (RECOVER-HSDP) uses the same setting and results are presented in Section A.3.

Setup. We pre-train a 7B-parameter LLaMA-style model on C4 [28] dataset under a 3D parallelism stack (TP×PP×DP), running on 512 A100 40GB GPUs, with the initial replica count $W_{\text{init}} = 64$ and



(a) (b)

Figure 7: Trajectory preservation under 256 GPU losses on a 512-GPU 3D-parallelism run. (a) The RECOVER-3D training loss curve matches the failure-free NCCL reference, with no spikes or measurable deviation. (b) RECOVER-3D improves per-GPU utilization along the failures due to versatile workload thus surpassing the failure-free reference.

grad-accum factor $G_{\text{init}} = 128$ given a global $B = W_{\text{init}} \cdot G_{\text{init}} = 8192$ microbatches per optimizer step. All failures are injected on a randomly chosen GPU via a randomly generated deterministic schedule by the simulator of Section C: 256 GPU losses are spread across the run, spaced at every 5 iterations, each at a randomly chosen step within its interval, to stress-test the system under frequent failure regimes. We deliberately inject all failures *during gradient synchronization*, which is the most difficult setting due to the existence of partially reduced gradients. We use grad-accum $= 8$ when comparing to *checkpoint-restart baseline* so that every component in the break-down of a failure recovery window would remain comparable. For fair comparisons, we sweep the baseline's checkpoint interval N from 2 to 64 iterations, and inject failures right at the middle of each interval. This reflects the practical consideration that checkpoint frequency should adapt to failure frequency. Our headline metric is *effective throughput* (tokens per second per alive GPU): unlike raw throughput, it factors out the shrinking world and directly measures resource utilization. Full evaluation details are in Section A.1.

Trajectory preservation: indistinguishable curves. Figure 7(a) plots the training loss of RECOVER-3D against the failure-free NCCL reference over the first 260 optimizer steps. Despite 256 GPU losses spread across the run, the two curves are visually indistinguishable, with no spikes, oscillations, or post-failure correction transients. This is the empirical evidence of the trajectory-preservation guarantee of Section F: every iteration commits the same number of microbatches to the optimizer, so the per-iteration gradient distribution is the same as the reference's, and the optimization trajectory inherits that equivalence. By contrast, prior keep-alive frameworks that allow the global batch to contract between failures and reconfigurations have been observed to produce loss fluctuations contingent on the failure schedule [30]; RECOVER intrinsically removes that contingency.

Quantitative equivalence and data coverage. Over the first 250 steps, RECOVER-3D consumes the same 8.39B tokens as the reference, of which 2.79B (33%) are drawn from survivors' streams in place of the shards held by the 32 lost replicas. No data is lost, only deferred: the displaced indices are determined by when and which replica failed, and can be scheduled into a subsequent run. As each replica's shard is an unbiased sample of the data mixture (Nanoset [33] randomly permutes the weighted global sample index before striding it across DP ranks), replacing a lost shard is equivalent to reshuffling the data stream under a different seed. Despite 33% of the tokens differing, the two runs remain statistically identical: the validation loss differs by at most 0.024 (0.36%) and ends 0.008 lower (Table 1). The same holds for RECOVER-HSDP (mean training-loss deviation 0.009, Section A.3) and under i.i.d. per-GPU failures (Section G.1).

Step	50	100	150	200	250
Failure-free reference (NCCL)	7.723	6.714	5.922	5.342	4.961
RECOVER-3D (256 GPUs lost)	7.721	6.689	5.923	5.342	4.953

Table 1: Validation loss on held-out data for the 512-GPU 3D-parallelism run of Figure 7(a).

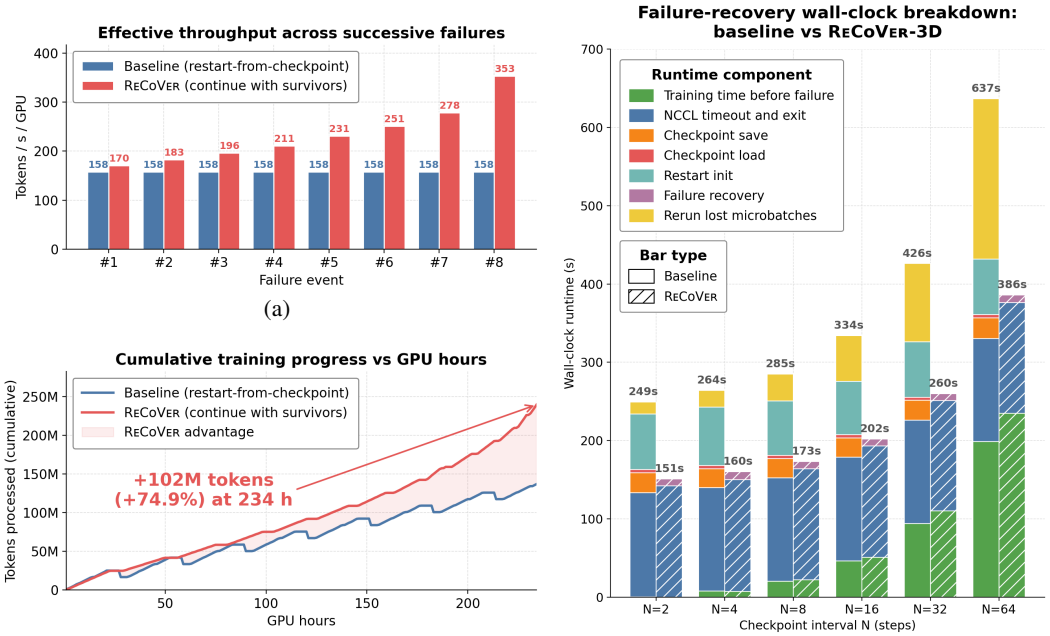


Figure 8: Cost comparison between ReCoVer-3D and restart-from-checkpoint. (a) Effective throughput across successive failures; ReCoVer-3D keeps increasing and enlarges the gap between baseline as the growing per-GPU workload amortizes the cross-replica all-reduce cost. (b) Cumulative training progress in tokens vs GPU-hours; ReCoVer-3D processes 74.9% more tokens at 234 GPU-hours. (c) Single-failure raw wall-clock breakdown swept over checkpoint interval N.

Effective throughput: drop, climb, exceed. Figure 7(b) shows effective throughput along the same run. Pre-failure, ReCoVer-3D matches the NCCL reference closely, where the gap comes from two parts: (1) we implemented ReCoVer without overlapping gradient synchronization with backward computation, while the baseline does; (2) the backend overhead of OpenMPI over NCCL. At the first failure, throughput drops sharply, which we speculate that it is caused by the cross-replica world reshape producing a topology that is less efficient for MPI backend. As more failures occur, gradient accumulation increases and amortizes the backend overhead by adding more compute per iteration, effectively raising per-GPU utilization. Concretely, with G the current grad-accum factor, s the tokens per microbatch, t_c the compute time per microbatch, and T_r the per-iteration cross-replica all-reduce cost, which stays constant as the gradient size is fixed by the model, the per-GPU effective throughput is

$$Gs/(Gt_c + T_r) = s/(t_c + T_r/G),$$

monotonically increasing in G toward the pure-compute rate s/t_c. The NCCL reference obeys the same form at its fixed G_{init}, so ReCoVer-3D overtakes it once

$$T_r^{\text{MPI}}/G_{\text{cur}} < T_r^{\text{NCCL}}/G_{\text{init}}.$$

In the high-failure regime the amortization dominates, and ReCoVer-3D's effective throughput climbs back and eventually exceeds the failure-free reference. This is not an artifact of the metric: each surviving GPU does *strictly more useful compute per unit time*, even as the optimization trajectory remains stochastically equivalent to the failure-free run.

Comparison with checkpoint-restart, across single and multiple failures. We now demonstrate how ReCoVer saves GPU hours over traditional checkpoint-restart approach on three fronts. Figure 8(a) reports effective throughput across successive failure intervals: the baseline is flat because every failure pays the same restart-and-rerun cost in expectation, while ReCoVer-3D's effective throughput *rises* monotonically as each new failure increases the compute workload for survivors and amplifies the amortization effect, which is then translated to strictly better resource utilization on every alive GPU. Figure 8(b) signifies this into cumulative training progress: at 234 GPU-hours, ReCoVer-3D has processed +102 M more tokens than the baseline, a 74.9% advantage that grows across the run and will keep growing until all replicas have failed. Figure 8(c) decomposes

a single recovery into its *raw* wall-clock components and sweeps the checkpoint interval N from 2 to 64 steps. The baseline's recovery cost grows with N because longer checkpoint intervals translate directly into more lost work to re-execute; RECOVER's cost is roughly flat across N because it never discards work. RECOVER even wins at the baseline's most favorable N ($N = 2$, extremely frequent checkpoints). Moreover, the restart cost (resource allocation, init, loading, first-step cold-start), though not significant at our test scale, is reported to be ~ 10 minutes on 100k production system [30] despite being optimized by industrial engineers. With the rapid scaling trend, the restart time alone will dominate baseline's recovery overhead and the gaps in Figure 8 will be even wider.

6 Discussion and Limitations

Dependence on ULFM and portability to NCCL. What RECOVER fundamentally relies on is not MPI itself, but an agreement among the survivors on which ranks have failed. Without such an agreement, survivors may post mismatched collectives and hang the job. NCCL can already rebuild a communicator over the survivors through `ncclCommShrink`, but it only repairs the communicator and leaves the decision on which ranks to exclude to the application. RECOVER makes this decision with `MPIX_Comm_agree`, whereas on NCCL it would require an out-of-band agreement mechanism. Since the middle and top layers of RECOVER never call MPI, they can be ported unchanged to any backend that provides both repair and agreement. Such a port would remove the backend overhead of OpenMPI over NCCL while preserving computational equivalence.

Replica as the unit of recovery, without rejoin. When a rank fails, RECOVER removes its entire replica rather than the failed rank alone. The remaining ranks of that replica no longer have enough GPUs for its original TP and PP configuration, and running them in a different configuration would affect all cross-replica collectives. Supporting such heterogeneous replicas is beyond the scope of this work, although our design does not preclude it. Moreover, removed replicas do not rejoin, so the number of replicas only decreases as failures accumulate. We deliberately let the failed replica exit as a whole in order to keep training progressing, and its resources are thus released immediately. In principle, rejoining can be implemented with `MPI_Comm_spawn`, similar to the rejoin path of FTAR [30]. However, spawned processes are not yet fully supported by current ULFM implementations. This is a limitation of the backend rather than of RECOVER, and our design is compatible with rejoining once the backend allows it.

Comparison with recent systems. ElasWave [15] makes communicator rebuilding cheap, but it recovers only at a step boundary and does not fully disclose how the survivors reach that boundary from an all-reduce in which a rank has failed. RECOVER detects the failure inside that all-reduce and preserves the survivors' intra-iteration progress, whereas ElasWave keeps only a per-step optimizer snapshot and thus re-runs the interrupted iteration. Morphlux [35] optically rewires a reserved spare accelerator into the position of the failed one and then restarts from the latest checkpoint, whereas RECOVER reserves no spare hardware and completes the failing iteration with the survivors.

7 Conclusion

RECOVER reframes resilient LLM pre-training as a *forward recovery* problem governed by a single invariant: each iteration commits gradients from the same number of microbatches as its failure-free reference, regardless of when or where failures occur. Its three-layer protocol enforces this invariant by locally containing failures at the communication layer, recovering partially reduced gradients within the failing iteration, and dynamically redistributing microbatch quotas across survivors. The design *extends* resilience across the full pre-training stack rather than only the communication primitive, remains *versatile* across both 3D and HSDP parallelism, and preserves *computational equivalence* to the failure-free trajectory without rollback, replay, or pre-allocated idle replicas. Compared to checkpoint-restart baselines, RECOVER achieves up to $2.23\times$ higher effective throughput after successive failures and processes 74.9% more tokens within 234 GPU-hours, with the advantage growing over longer training runs. Given modern contemporary HPC systems of +100k-GPUs that are used for pre-training, RECOVER will not suffer from the increasing restart overhead that becomes a major bottleneck for traditional checkpoint-restart approaches, and remains a resource-efficient resilience solution. In future work, we will refine RECOVER to recycle the surviving ranks of failed replicas (currently discarded) in order to further improve system throughput. Also, we will explore how to allow fresh replicas to rejoin dynamically in order to better control the training progress.

Acknowledgments and Disclosure of Funding

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Artificial Intelligence for Science program, under contracts DE-SC0025390 and DE-AC02-06CH11357.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP0030039, as well as NERSC award ALCC-ERCAP0031379.

References

[1] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[2] W. Bland, H. Lu, S. Seo, and P. Balaji. Lessons learned implementing user-level failure mitigation in mpich. In *2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing*, pages 1123–1126. IEEE, 2015.

[3] A. Bouteiller, G. Bosilca, and J. J. Dongarra. Plan b: Interruption of ongoing mpi operations to support failure recovery. In *Proceedings of the 22nd European MPI Users' Group Meeting*, pages 1–9, 2015.

[4] S. Dash, I. R. Lyngaas, J. Yin, X. Wang, R. Egele, J. A. Ellis, M. Maiterth, G. Cong, F. Wang, and P. Balaprakash. Optimizing distributed training on frontier for large language models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–11. Prometeus GmbH, 2024.

[5] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram. Check-N-Run: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[6] S. Gandhi, M. Zhao, A. Skiadopoulos, and C. Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 211–228, 2024.

[7] M. Gooding. xai targets one million gpus for colossus supercomputer in memphis, 2024.

[8] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[9] S. Hasan. Scaling llama4 training to 100k, 2026.

[10] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, et al. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, 2024.

[11] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[12] I. Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[13] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[14] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, et al. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[15] X. Kang, G. Xiang, Y. Wang, H. Zhang, Y. Fang, Y. Zhou, Z. Tang, Y. Lv, E. Maman, M. Wasserman, et al. Elaswave: An elastic-native system for scalable hybrid-parallel training. *arXiv preprint arXiv:2510.00606*, 2025.

[16] A. Kokolis, M. Kuchnik, J. Hoffman, A. Kumar, P. Malani, F. Ma, Z. DeVito, S. Sengupta, K. Saladi, and C.-J. Wu. Revisiting reliability in large-scale machine learning research clusters. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1259–1274. IEEE, 2025.

[17] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski. Evaluating user-level fault tolerance for mpi applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 57–62, 2014.

[18] J. Lee, Z. Chen, X. He, R. Underwood, B. Nicolae, F. Cappello, X. Lu, S. Di, and Z. Zhang. Spare: Stacked parallelism with adaptive reordering for fault-tolerant llm pretraining systems with 100k+ gpus. *arXiv preprint arXiv:2603.00357*, 2026.

[19] J. Li, G. Bosilca, A. Bouteiller, and B. Nicolae. Elastic deep learning through resilient collective operations. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 44–50, 2023.

[20] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi. Fault tolerance of mpi applications in exascale systems: The ulfm solution. *Future Generation Computer Systems*, 106:467–481, 2020.

[21] A. Maurya, M. M. Rafique, F. Cappello, and B. Nicolae. Datastates-llm: Scalable checkpointing for transformer models using composable state providers. *arXiv preprint arXiv:2601.16956*, 2026.

[22] A. Maurya, R. Underwood, M. M. Rafique, F. Cappello, and B. Nicolae. Datastates-llm: Lazy asynchronous checkpointing for large language models. In *Proceedings of the 33rd international symposium on high-performance parallel and distributed computing*, pages 227–239, 2024.

[23] J. Mohan, A. Phanishayee, and V. Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[24] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[25] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al. 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*, pages 1–15, 2021.

[26] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello. Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 911–920. IEEE, 2019.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[28] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[29] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pages 1–16. IEEE, 2020.

[30] O. Salpekar, R. Varma, K. Yu, V. Ivanov, Y. Wang, A. Sharif, M. Si, S. Xu, F. Tian, S. Zheng, et al. Training llms with fault tolerant hsdp on 100,000 gpus. *arXiv preprint arXiv:2602.00277*, 2026.

[31] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[32] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[33] N. Tazi, F. Mom, H. Zhao, P. Nguyen, M. Mekkouri, L. Werra, and T. Wolf. The ultra-scale playbook: Training llms on gpu clusters. 2025. *URl: https://huggingface.co/spaces/nanotron/ultrascaleplaybook*, 2025.

[34] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[35] A. Vijaya Kumar, E. Ding, A. Devraj, D. Bunandar, and R. Singh. Reconfigurable torus fabrics for multi-tenant ml. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1547–1565, 2026.

[36] B. Wan, M. Han, Y. Sheng, Y. Peng, H. Lin, M. Zhang, Z. Lai, M. Yu, J. Zhang, Z. Song, et al. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, 2025.

[37] B. Wan, G. Liu, Z. Song, J. Wang, Y. Zhang, G. Sheng, S. Wang, H. Wei, C. Wang, W. Lou, et al. Robust llm training infrastructure at bytedance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 186–203, 2025.

[38] W. Wang, N. Yu, S. Xiong, and Z. Liu. Reliable and resilient collective communication library for llm training and serving. *arXiv preprint arXiv:2512.25059*, 2025.

[39] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. E. Ng, and Y. Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[40] Z. Wang, Z. Liu, R. Zhang, A. Maurya, P. Hovland, B. Nicolae, F. Cappello, and Z. Zhang. Boost: Bottleneck-optimized scalable training framework for low-rank large language models. *arXiv preprint arXiv:2512.12131*, 2025.

[41] xAI. Grok 4, 2025.

[42] Y. Xiong, Y. Jiang, Z. Yang, L. Qu, G. Zhao, S. Liu, D. Zhong, B. Pinzur, J. Zhang, Y. Wang, et al. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 835–850, 2024.

[43] H. Zhang, D. Morwani, N. Vyas, J. Wu, D. Zou, U. Ghai, D. Foster, and S. Kakade. How does critical batch size scale in pre-training? *arXiv preprint arXiv:2410.21676*, 2024.

[44] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

A Additional Evaluations and Details

This appendix provides additional evaluation details and the results for RECOVER-HSDP that cannot be included in the main text due to the tight page limit.

A.1 General Details of All RECOVER Evaluations

Testbed. All evaluations are conducted on up to 128 nodes of Polaris at the Argonne Leadership Computing Facility (ALCF). Each node is equipped with 4 NVIDIA 40GB A100 GPUs. Intra-node communication uses NVLink, and inter-node communication uses a high-speed Slingshot.

Software Stack. We use CUDA-aware OpenMPI 5.0.8 [2] as the communication backend for cross-replica gradient reduction. It is achieved by extending the open-source PyTorch [3] a customized process group backend (see Section C). For 3D parallelism, we build upon Nanotron [33], an open-source high-performance pre-training framework developed by Huggingface [4]. For HSDP, we directly use FSDP1+`HYBRID_SHARD` [44] that PyTorch ships, and a self-implemented training loop that mirrors Nanotron's logic.

General Setup. For comparisons with failure-free references, we experiment on 128 nodes, i.e., 512 GPUs, to demonstrate RECOVER's scalability and robustness. Among these experiments, 256 GPUs are lost spread across the run. For comparisons with checkpoint-restart baselines, we experiment on 32 nodes, i.e., 128 GPUs, for faster turnaround time as 128-node jobs are very difficult to be queued. We remark that benchmarking against checkpoint approaches at a smaller scale actually favors baselines: the checkpoint-restart stall time (resource allocation, initialization, checkpoint save/load, first step cold start, rerun lost progress) increases dramatically as the system scales up, and is projected to be even longer than failure interval at hyper-scales [30, 18], i.e., 10 min stall time for 100k system, but MTBF is projected to be ~ 5 min for 600k system, while RECOVER does not suffer from these costs.

Failure Injection. We use the failure simulator described in Section C plus NCCL watchdog timeout to abort the entire program in checkpoint-restart baselines, and to abort the faulty replicas in RECOVER. All failures are injected precisely during the gradient synchronization, which is the most challenging setting, especially for examining trajectory preservation. Every failure will result in partially reduced gradients, and RECOVER needs to correctly restore the affected ones else the curve might drift away.

For the single-failure runtime breakdown experiment, we assume a checkpoint interval of N, meaning that the baseline saves checkpoints every N steps. We then inject a failure at step $1.5N$, corresponding to the expected midpoint of a checkpoint interval under an independent failure assumption. For example, when the checkpoint interval is 8, the failure is injected at step 12.

For the multi-failure experiment, we again use checkpoint interval N, and inject failures starting at step $1.5N$, then every N steps thereafter, i.e., at $1.5N, 2.5N, \ldots$.

Profiling. For the single-failure experiment, in the baseline setting, we treat the first N steps as warmup and exclude them from measurement. We start recording after step N, immediately before checkpoint save, denoted as timestamp A. The checkpoint save completion time is denoted as B. The job then runs for another $N/2$ steps, and the failure is injected at step $1.5N$, denoted as C. We record the time when the first run fully exits as D, which is also the start time of the recovery launch. The checkpoint load start and finish times are denoted as E and F, respectively. The resumed training start time is denoted as G. After rerunning another $N/2$ steps, the completion time of step $1.5N$ is denoted as H.

The runtime breakdown is then computed as follows:

- checkpoint save time = $B - A$
- normal run time before failure = $C - B$
- failure handling time = $D - C$

[2] https://docs.open-mpi.org/en/v5.0.x/.

[3] We install a developer version PyTorch (2.9.0, forked in July 2025) from source with OpenMPI installation to enable MPI-aware PyTorch.

[4] https://huggingface.co/nanotron.

- checkpoint load time = $F - E$
- restart initialization time = $G - D - (F - E)$
- rerun time = $H - G$

For RECOVER-3D, we start recording at step N, then record the failure injection time, the time when recovery completes, and the time when step $1.5N$ completes.

For the multi-failure experiment, we measure the runtime between each pair of consecutive failures. For the GPU-hour based metric, we sample the current completed iteration at each GPU-hour boundary and use it to compute the total number of processed tokens.

Metric Definitions. For the single-failure experiment, we report the *raw* runtime breakdown that is defined by the runtime of each event described above. For the multi-failure experiment, we measure the runtime between consecutive failures.

Effective throughput is defined as:

$$\text{Effective Throughput} = \frac{\text{Processed Tokens}}{\text{Runtime} \times \text{Alive GPUs}}.$$

Under this definition, the baseline remains approximately flat across failures because it restores the same configuration after each restart and therefore makes the same progress per failure interval. In contrast, RECOVER operates with fewer and fewer surviving GPUs after successive failures.

For the processed-tokens versus GPU-hours metric, we record the completed iteration at each GPU-hour boundary and convert it to processed tokens. Together, these two metrics characterize resource efficiency and show that RECOVER can significantly improve GPU utilization under repeated failures.

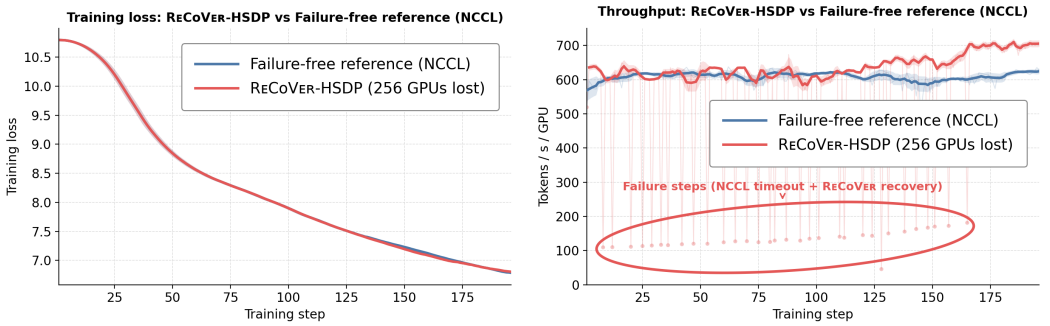
A.2 Additional Details of RECOVER-3D Experiments

Training Configuration. We train a LLaMA-style 7B model with micro-batch size $= 1$, sequence length $= 4096$, and BF16 training with gradient accumulation/synchronization in FP32. For the parallelization strategy, we use TP$= 4$ within each node, PP$= 2$. For comparisons with the failure-free reference, we use gradient accumulation $= 128$ and DP$= 64$; for comparisons with checkpoint-restart, we use accumulation $= 8$ and DP$= 16$. The choice of smaller gradient accumulation for the latter is for the wall-clock breakdown shown in Figure 8(c), so that each component remains comparable. As we don't have access to an actual 100k system, this is due to practical consideration for showing every aspect of the checkpoint-restart strategy is non-trivial.

A.3 Evaluations of RECOVER-HSDP

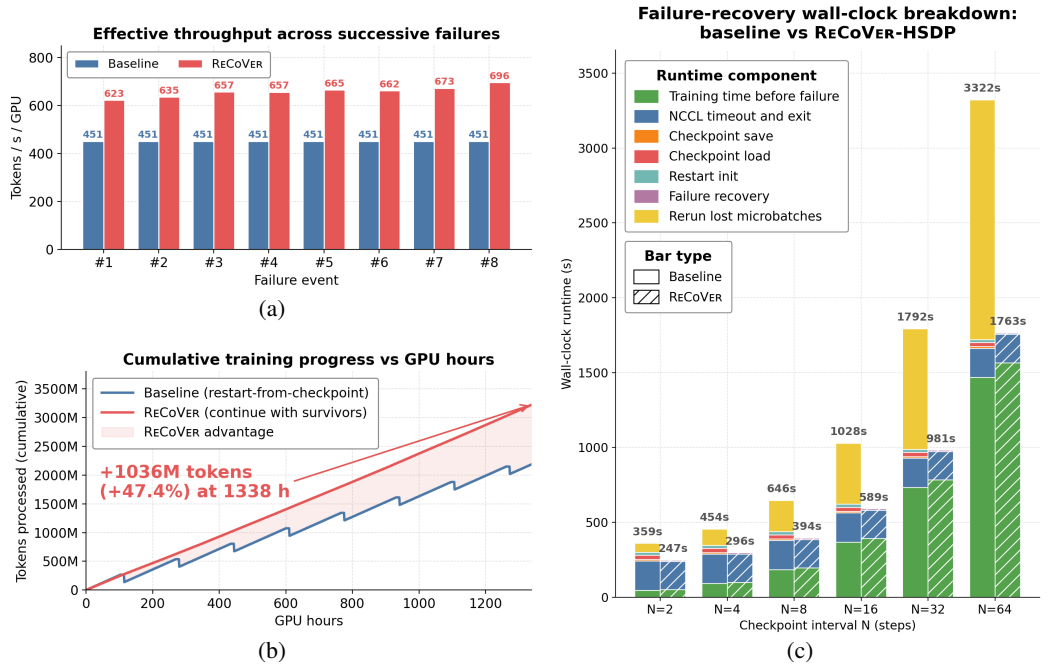
Training Configuration. We train a LLaMA-style 1B model with micro-batch size $= 1$, sequence length $= 4096$, gradient accumulation $= 16$, HSDP shard size $= 8$, and BF16 training with gradient accumulation/synchronization in FP32. For comparison with the failure-free reference, we have 64 replicas, while for the comparison with checkpoint-restart baseline, we have 16 replicas. The number of replica is deliberately chosen to remain the same as 3D parallelism's setting so we can simulate same number of replica failure. Consequently, the model size is reduced from 7B to 1B, as 7B would not fit in 8 40GB GPUs under HSDP, unless increasing the sharding size.

Evaluation Results. Similar to the results of RECOVER-3D shown in Section 5, we present the comparison of RECOVER-HSDP against a failure-free reference and a standard checkpoint-restart baseline. In Figure 9(a), we see that RECOVER-HSDP again produces a loss curve that is indistinguishable to its failure-free run, and maintains an effective throughput that matches the NCCL reference and increases as failures keep happening. These results are mostly identical to RECOVER-3D case, except RECOVER-HSDP is not even slower at the beginning, possibly due to that each local shard of the model is smaller in 1B case (as we deliberately chose replica size to be the same, i.e., 8 GPUs a replica, so each shard is effectively smaller) so that OpenMPI's backend overhead is less pronounced. In Figure 10(c), we see the gap between checkpoint baseline and RECOVER-HSDP remains the same ratio as the ones in RECOVER-3D, except here the checkpoint-related overhead is less comparable to other components as that each HSDP iteration is longer than it was in 3D parallelism's case. And similarly, 10(a) and Figure 10(b) show similar benefit over checkpoint baselines. The gap is slightly smaller than in RECOVER-3D, which is reasonable as our improvement



(a) (b)

Figure 9: Trajectory preservation under 256 GPU losses on a 512-GPU HSDP run. (a) The ReCoVer-HSDP training loss curve matches the failure-free NCCL reference throughout the run, with no spikes or measurable deviation. (b) The corresponding effective throughput: ReCoVer-HSDP matches NCCL closely, and surpasses it after successive failures as ReCoVer increases per-iteration workload and improves per-GPU utilization.



(a) (b) (c)

Figure 10: Cost comparison between ReCoVer-HSDP and restart-from-checkpoint. (a) Effective throughput across successive failures; ReCoVer-HSDP is consistently higher than checkpoint baseline. (b) Cumulative training progress in tokens vs GPU-hours; ReCoVer-HSDP processes 47.4% more tokens at 1338 GPU-hours. (c) Single-failure raw wall-clock breakdown swept over checkpoint interval N. ReCoVer wins even at baseline's most favorable setting (i.e., $N = 2$).

on resource utilization comes from pushing the hardware toward compute-bound regime. In 3D parallelism, communications are highly structural: high-frequency and high-volumes ones, such as the TP all-reduce is restrained to intra-node, while inter-node communication only handles PP send/recv and gradient all-reduce, which are much lighter than HSDP's frequent all-gather/reduce-scatter, that happens on every parameter's every forward/backward across all devices in each replica, which span across multiple nodes. On a system that is not deliberately designed/optimized for inter-node GPU communication, HSDP is much less scalable and efficient than 3D parallelism, which can be reflected by comparing Figure 8(a) and Figure 10(a): despite training a smaller model (1B), HSDP's effective throughput is only $\sim 1/3$ of 3D parallelism's. Therefore, we remark that this gap difference is due to

HSDP's inner structure and its interaction with our testbed hardware, not caused by RECOVER's design or implementation.

B Extended Terminology

Table 2 consolidates the terms used throughout the paper.

Term	Description
Microbatch	one forward-backward pass done by a replica.
Gradient synchronization	gradient all-reduce across the replicas
Gradient accumulation	gradient aggregation of G microbatches on each replica before gradient synchronization.
Iteration	the full cycle of gradient accumulation, synchronization, model update; model parameters are updated exactly once per iteration.
Replica	One complete copy of the model.
Microbatch size	Number of tokens a replica processes in one microbatch.
Global batch B	Number of microbatches contributed to an iteration, summed across replicas.
$W_{\text{init}}, G_{\text{init}}$	Initial replica count and initial grad-accum.
Major / minor	Replica roles assigned by versatile-workload policy; see Section 4.3.
Major-spare / minor-spare	Replica roles assigned by versatile-workload policy; see Section 4.3.
Global rank	A rank's index in the world communicator.
DP rank	A replica's index along the data-parallel dimension.
TP rank, PP rank	A rank's index inside its replica's tensor / pipeline group.
MP rank	A rank's index within its replica (across TP and PP together).
Shard rank	A rank's index inside its HSDP shard group.
Policy boundary	The moment where a failure has exhausted spares and extra microbatches must be produced at the current iteration, and a new grad-accum must be adopted for the future iterations.
Policy boundary step	The step to perform the extra microbatches when reaching policy boundary.

Table 2: Consolidated terminology used in the main text and appendices.

C Implementation

This appendix describes how RECOVER's three protocol layers (Section 4) are realized in open-source PyTorch [27], and Nanotron [33]. The implementation is laid out as a PyTorch-native ULFM communication backend (`ProcessGroupULFM`, written in C++) plus a Python control plane (`ulfm_collectives/`) that turns the fault-tolerant collective into a fault-tolerant optimizer step. We integrate the control plane with three pre-training stacks — a flat DDP loop, a HSDP (FSDP1 [44] with `HYBRID_SHARD`) loop, and a Nanotron 3D(+EP) loop.

`ProcessGroupULFM`: the fault-tolerant backend. `ProcessGroupULFM` subclasses PyTorch's `ProcessGroup` and is registered under the backend name `"ulfm"`; RECOVER selects this backend for the cross-replica process group PG_{cross} (the data-parallel group in 3D parallelism; the FSDP *replicate* group in HSDP). The backend exposes exactly the four ULFM phases of Section 4.1 as two PyTorch-native collectives: ULFM_ALLREDUCE wraps an in-place sum all-reduce with the four-phase *detect–repair–record–reduce* sequence, and ULFM_CONSENSUS wraps the first three phases as a barrier-like collective for failure view synchronization. Internally, both collectives drive the standard ULFM primitives `MPIX_Comm_agree`, `MPIX_Comm_revoke`, and `MPIX_Comm_shrink` on the underlying MPI communicator, and on every successful repair bump a monotone integer `worldEpoch()` that the Python control plane stamps onto each gradient-bucket snapshot to drive the world-epoch classification of Algorithm 5. Each call returns a `WorkULFM` future that carries failure metadata — `has_failures()`, `get_failed_ranks()`, the survivor census, and (when versatile-workload tracking is enabled) per-role failure counts and contribution counters — so the Python hook can inspect a single work object after `.wait()` and drive all downstream policy logic from it. A pg-level `set_quiesce(true/false)` latch lets the control plane short-circuit any further bucket all-reduces to a pre-resolved future once a failure has been observed in the window, avoiding doing meaningless work that are meant to be restored even succeeded.

The same backend also carries the versatile-workload state for the top protocol layer: per-rank atomic flags (`is_minor_`, `is_spare_`, `is_boundary_minor_`) plus per-role atomic counters (`num_major_procs_`, `num_minor_procs_`, `num_major_spare_procs_`, `num_minor_spare_procs_`, `num_boundary_minor_procs_`) and per-rank contribution counters (`contributed_`, `target_contribution_`, plus a separate `boundary_*` pair used during a policy boundary step). When a failure is detected, two helpers run *within* the allreduce: `detect_and_recover_failures` (i) lists the failed ranks, (ii) repairs the communicator and bumps the epoch, and `record_and_handling_failure` (iii) re-censuses survivors, (iv) computes per-role failure counts, (v) evaluates the policy-boundary predicate (major-failure with no major-spare *or* minor-failure with no minor-spare), (vi) latches the `atPolicyBoundary_` flag if true, and (vii) when no boundary is crossed, runs `elect_promotion` to atomically promote a spare into the vacated role. Everything is then attached to the `WorkULFM` that returns to Python, so W_{cur}, C_{cur}, the per-role failure counts, and the boundary verdict arrive at the policy layer in one collectively agreed package.

`StepTxnOrchestrator`**: per-iteration transaction state.** The orchestrator owns the iteration-local state: the list of pre-reduce bucket snapshots $\{(b, S(b), \epsilon(b), \text{idx}(b))\}$, the set of buckets that have reduced cleanly under the current epoch, the deferred bucket queue that needs to be drained after backward computation finalizes, and the current restoration plan. It exposes a single unified entry point, `handle_work_completion(work, bucket_index)`, that every future of `WorkULFM` invokes upon `.wait()`: on success it marks the bucket reduced under the current epoch; on failure it (i) unpacks `FailureStats` and `RankTypeCounts` from the `WorkULFM`, (ii) packages them with its own progress tracker (current microbatch index, total in-window, world epoch, current rank/size) into a `FailureEvent`, (iii) consults the policy, (iv) latches `at_policy_boundary` and the restore mode, and (v) either quiesces the pg or proceeds based on the policy decision. Two restore implementations live on the orchestrator: `restore_gradients_non_blocking()` schedules per-snapshot `view.copy_(snap)` on a dedicated CUDA stream and records an event that the trainer waits on right before the next backward, so the rewind is overlapped with the boundary step's forward; `restore_gradients_blocking()` performs the rewind plus a re-issued ULFM_ALLREDUCE on every stale bucket synchronously before the optimizer step, with a guarded retry path for the rare case in which the re-reduction itself hits a failure: if the failure is a non-boundary-crossing failure, retry the re-reduction; if it reaches policy boundary, break from the loop and enters the policy boundary step logic. After the optimizer step commits, `after_successful_commit()` drives the policy advance: `policy.advance_policy()` returns the new $(n_{\text{maj}}, n_{\text{min}}, n_{\text{ms}}, n_{\text{mi}})$ layout (Algorithm 7), which is then pushed into the C++ backend via `set_major_minor_split_with_spares` and `update_rank_type_counts`, the `at_policy_boundary` latch is cleared, and `set_target_contribution` primes the per-role microbatch quotas for the next iteration.

TrainingManager: the microbatch state machine. The TrainingManager is the only component that touches the PyTorch module and the optimizer. Across all three integrations it preserves the same per-microbatch state machine: at $m = 0$ in the window it runs `on_iteration_start()` and zeros the gradient buffers; on each microbatch, if the current restore plan is NON-BLOCKING (an extended boundary pass) it kicks off the asynchronous rewind onto a dedicated CUDA stream so it overlaps with the upcoming forward; before the backward of the first extended-pass microbatch, it calls `wait_restore_before_backward()` to synchronize the rewind; on every microbatch it consults `pg.should_contribute()` and zeros the loss for that microbatch when the rank has already met its quota for the iteration; after synchronization it normalizes the accumulated gradient by the target global batch (`effective_batch_size` = $W_{\text{init}} \cdot G_{\text{init}}$, constant under `StaticWorldPolicy`), runs the optimizer step, and calls `after_successful_commit()` so the policy can advance the role layout (Algorithm 7) for the next iteration. The three training-stack integrations described below differ only in (a) which comm hook is registered on the model and (b) which entity drives the outer per-iteration recovery loop; the shared microbatch mechanics, the orchestrator, and the policy machinery are unchanged.

Path 1 — DDP-only, immediate hook. The flat-DDP path is the simplest and the only one that submits MPI work *inside* backward. `ULFMTrainingManager` wraps the model in standard PyTorch `DistributedDataParallel` on the ULFM `WORLD` process group and registers `create_ulfm_recovery_hook` as the comm hook. Each backward triggers the hook once per gradient bucket: the hook snapshots the bucket via `orch.on_bucket_snapshot`, posts

ULFM_ALLREDUCE on $PG_{\text{cross}} = \texttt{WORLD}$, and on the future's `then(...)` callback routes the resulting `WorkULFM` through `orch.handle_work_completion`. Because the bucket schedule is a flat list with no pipeline or shard collectives running concurrently on the same device, blocking inside the hook does not interfere with any other collective on the same rank, so the immediate path is preferred: The hook awaits ULFM_ALLREDUCE inline, so any failure is reported on the exact bucket whose reduction it interrupted, and the orchestrator runs its restore-plan logic before `train_step` returns — the trainer's outer loop simply calls `train_step` again for the next microbatch, with no recovery-aware control flow of its own. This path is used in the DDP-only single-replica-per-rank configuration as a reference implementation of the protocol mechanics; it is not the path used in experiments of Section 5.

Path 2 — 3D parallelism on Nanotron, fp32-deferred hook. RECOVER-3D extends Nanotron with a `ULFMParallelContext` that places the data-parallel group on the ULFM backend and TP/PP/EP on NCCL with extended async-error-handling timeouts. `NanotronULFMTrainingManager` binds the orchestrator to the DP process group and, when Nanotron's `accumulate_grad_in_fp32` is enabled (the common case for LLM pre-training), registers `create_ulfm_fp32_deferred_hook` on the inner DDP. The hook does three things in one place: (i) accumulates the bucket's bf16 gradients into Nanotron's contiguous fp32 grad accumulator (`_contiguous_fp32_grad_buffer`) unconditionally, so that local accumulation is preserved even if the cross-replica pg is quiesced; (ii) derives the contiguous slice of the fp32 buffer covering this bucket's parameters (asserting gap-freeness, which holds under DDP's reverse-registration bucketing) and snapshots that slice for failure rollback; and (iii) queues the same slice *view* on the orchestrator's deferred-bucket queue and returns a pre-resolved future, so DDP's `finalize_backward` never blocks on ULFM logic, and therefore the following PP `send/recv` on activation gradients won't timeout and kill the ranks in healthy replicas. The actual cross-replica all-reduce lands later, when the trainer's outer loop calls `fire_deferred_allreduces`; because the queued buffer is a view into the fp32 accumulator's storage, the reduction lands in place and no scatter-back into per-parameter grads is needed. The outer per-iteration recovery loop in `trainer_ulfm.py` then barriers on `mp_pg` (the union of TP+PP+EP group) as the replica-consistency gate to ensure faulty replica's members are killed cleanly, run a ULFM_CONSENSUS on `dp_pg` to ensure consistent post-failure view across intra-replica ranks in healthy replicas, and dispatch on the resulting restore mode — SKIP commits the optimizer step, NON-BLOCKING starts the async rewind and re-enters the loop with the boundary-extension microbatch count, and BLOCKING runs a synchronous rewind-and-re-reduce that optionally crosses into the non-blocking branch if the re-reduce itself trips a policy boundary.

Path 3 — HSDP, deferred hook. RECOVER-HSDP wraps the model with FSDP1 in `HYBRID_SHARD` mode, with the intra-replica `shard_pg` on NCCL and the cross-replica `replicate_pg` on RECOVER's ULFM backend. `HSDPULFMTrainingManager` binds the orchestrator to `replicate_pg` and registers `create_ulfm_hsdp_hook` on the FSDP-wrapped model. Critically, the hook is also *deferred*: when FSDP fires it on the sync microstep with the unit's `flat_param._saved_grad_shard`, the hook snapshots the shard accumulator, enqueues the buffer reference on the orchestrator's deferred-bucket queue, and returns a pre-resolved future. Similarly, the non-overlapping scheme is to prevent a cross-replica ULFM logic on `replicate_pg` co-occurring with intra-replica reduce-scatter and all-gather collectives on `shard_pg`, where the NCCL watchdog on `shard_pg` could time out on healthy replicas and terminated the very ranks supposed to drive recovery. By deferring the cross-replica MPI work to after the inner microbatch loop has fully returned, the intra-replica NCCL traffic on `shard_pg` runs to completion on a clean schedule. To preserve the invariant of Equation (1) under this change, `HSDPULFMTrainingManager` also strips FSDP's default replicate-axis grad post-divide so that the gradient divisor is owned solely by the manager's `_get_grad_div_factor` and stays constant at $W_{\text{init}} \cdot G_{\text{init}}$ regardless of which replicas survive. The outer per-iteration recovery loop is then driven by `main_hsdp.py`: run the inner microbatch loop, drain the deferred queue via `fire_cross_replica_allreduces` (which calls `txn.fire_deferred_allreduces`), barrier over `shard_pg`'s ranks as the replica-consistency gate, run a ULFM_CONSENSUS on `replicate_pg` to ensure consistent post-failure view across intra-shard ranks, and dispatch on the resulting restore mode, same as Path 2.

Policies. The Python `FaultTolerancePolicy` hierarchy contains two concrete classes. `StaticWorldPolicy`, the policy described in Section 4.3 and used in all RECOVER experiments, implements the in-iteration boundary handling of Algorithm 6 (the boundary-step computation that

picks G_{ext} as the smallest integer with $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} \geq B$ and the corresponding boundary-minor split) and the post-boundary steady-state advance of Algorithm 7. `AdaptiveWorldPolicy` is the strawman of Algorithm 8: on every failure it returns REPAIR-AND-CONTINUE with a blocking restore and no boundary handling, so the global batch shrinks with the world. We keep it in the codebase so the same training stack and the same collective backend can be re-used as an elasticity-only baseline, which isolates the contribution of the versatile-workload layer.

Replica-consistency gate. All three paths use the gate introduced in Section 4.4 to abort a replica as a unit when one or more of its intra-replica ranks have failed. Path 1 trivially satisfies the gate because every rank is its own replica. Path 2 barriers on `mp_pg` (the union of TP/PP/EP). Path 3 barriers over `shard_pg`'s ranks.

Failure simulator. For reproducible evaluation, the framework ships a deterministic failure simulator that either generates or consumes a YAML schedule of (step, replica id, local rank, location) entries. At each injection point (forward, backward, gradient synchronization, optimizer), every rank checks whether its scheduled location matches the current one; if it does, the rank issues `os.kill(os.getpid(), SIGKILL)` to faithfully simulate a crash failure. The schedule is a pure function of (parallelism spec, seed, count, step range, location weights), so every rank generates the same schedule at startup without any cross-rank broadcast.

D Additional Algorithmic Details

This appendix expands the six black-box helpers called from the main-text Algorithm 1, in the order they fire during an iteration: ULFM_ALLREDUCE (Algorithm 2) on every gradient bucket of the last microbatch; ULFM_CONSENSUS (Algorithm 3) once after the bucket loop, as the iteration's final cross-replica gate; HANDLE_WORK_FAILURE (Algorithm 4) and GRADIENT_RESTORATION (Algorithm 5) if any of the ULFM collectives above flagged a failure; POLICY_ADJUSTMENT (Algorithm 6) immediately after, to decide whether the iteration must run extra microbatches; and finally POLICY_ADVANCEMENT (Algorithm 7) once a boundary iteration has committed, to install the next iteration's role layout. We give each helper at the level of *what it does and why*, and link each to its concrete implementation in Section C.

Notation and shared state. We reuse the symbols of Section 4. PG_{cross} is the cross-replica process group on the ULFM-aware backend; PG_{intra} is each replica's intra-replica group on NCCL. Each PG carries a monotone integer ϵ_{cur} (its *world epoch*) that the backend increments on every successful repair. When the bucket loop snapshots a gradient bucket b into $S(b)$, it tags the snapshot with the ϵ_{cur} in force at the time; we call b *stale* if $\epsilon(b) < \epsilon_{\text{cur}}$ — i.e. its most recent reduction (if any) was issued under a now-shrunk membership and would carry the wrong weight if mixed with current-epoch reductions in the iteration sum. $W_{\text{cur}}, G_{\text{cur}}, B$ are the current replica count, current grad-accum factor, and target global batch from Section 4.3; $C_{\text{cur}} = \sum_r C_r(t)$ is the running contribution count.

The `Work` object returned by Algorithm 2 and Algorithm 3 carries the reduction's result (when one occurred) plus a *failure record* with three fields that the remaining helpers consume:

- `role_counts` $= (n_{\text{maj}}, n_{\text{min}}, n_{\text{ms}}, n_{\text{mi}}, n_{\text{bm}})$: the post-failure (and, if applicable, post-promotion) population of each replica role on PG_{cross} — majors, minors, major-spares, minor-spares, and boundary-minors. The policy reads this both to detect whether a spare absorbed the failure (one of $n_{\text{ms}}, n_{\text{mi}}$ shrank by one relative to the pre-failure count) and to know how many survivors of each role remain.
- `contrib` $= C_{\text{cur}}$: the microbatches that survivors have *already* finished in this iteration at the moment of failure. The policy uses it at a boundary to size the extension: G_{ext} is chosen as the smallest integer with $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} \geq B$, so without an honest C_{cur} the extension would either overshoot or undershoot B.
- `at_boundary`: **true** iff a major failed with no major-spare or a minor failed with no minor-spare — equivalently, the failure could not be absorbed by a spare and the iteration must be extended.

D.1 ULFM-guarded collectives

Algorithm 2 is a fault-aware sum all-reduce: it rarely reduces under a failed membership, and it never crashes on a failed rank. The four phases of Section 4.1 divide labour cleanly: *Detect* is a cheap probe placed *before* any data motion, so a stale membership is caught before a reduction is posted; *Repair* shrinks PG_{cross} to the survivors and bumps ϵ_{cur} so downstream code can tell what was reduced under what; *Record* produces the failure record described above with the guarantee that every survivor walks away with the same record (one all-reduce on the per-replica role flags pins down `role_counts` and `contrib`; `at_boundary` is a deterministic local function of `role_counts` and is therefore agreed by construction; spare promotion, when it fires, is a small collective election so the post-promotion `role_counts` is the same on every survivor); and *Reduce* is the actual data motion, which runs only if Detect passed. The implementation lives in `ProcessGroupULFM` (Section C); the body of *Record* corresponds directly to the C++ helper `record_and_handling_failure` described there.

Algorithm 2 ULFM_ALLREDUCE (t, PG): fault-aware sum all-reduce.

Require: Tensor t, fault-tolerant process group PG.
Ensure: `Work` carrying the result and a failure record.
if PG was quiesced by an earlier failure this iteration **then return** a no-op `Work`.
Detect: probe PG for any rank that has failed since the last call on PG.
if a failure was detected **then**
    Repair: revoke pending operations on PG, shrink PG to the survivors, and increment $\epsilon_{\text{cur}}(PG)$.
    Record (collective): census the per-replica role flags via one all-reduce on PG to obtain `role_counts` and `contrib`; set `at_boundary` ← (a major died with no major-spare) **or** (a minor died with no minor-spare); if **not** `at_boundary`, run a small election on PG that promotes one major-/minor-spare into the vacated role and re-census so `role_counts` reflects the new layout; attach the failure record to `Work`.
    return `Work` (no reduction performed).
end if
Reduce: if this rank is a spare and `at_boundary` is **false**, set $t \leftarrow 0$; then sum-reduce t in place over PG.
return `Work` marked successful.

Algorithm 3 is the same primitive without the data motion of phase 4. Algorithm 1 fires it *once per iteration*, after every per-bucket ULFM_ALLREDUCE has returned and after the intra-replica barrier on PG_{intra}. It serves an important purpose at this position: it converts any asymmetric failure outcome from the bucket loop — one rank's ULFM_ALLREDUCE returning failure while a peer's returned success — into a globally agreed verdict, so every survivor sees the same failures together. The implementation shares the entry point of ULFM_ALLREDUCE in `ProcessGroupULFM` (Section C).

Algorithm 3 ULFM_CONSENSUS (PG): fault-aware barrier with no data motion.

Require: Fault-tolerant process group PG.
Ensure: `Work` carrying a failure record (or success).
Run *Detect*, *Repair*, and *Record* from Algorithm 2 on PG.
return `Work`.

D.2 Failure handling

Algorithm 4 bridges the collective backend and the policy: it consumes the failure record produced by Record (Algorithm 2), asks the policy what to do (POLICY_ADJUSTMENT), and installs the answer as state visible to the rest of this iteration — a latched `restore_mode` that GRADIENT_RESTORATION will read, a quiesce on PG_{cross} that short-circuits any further bucket all-reduce in this iteration to a no-op (those buckets will be rolled back anyway), and the boundary-minor split that controls which replicas contribute fewer microbatches on the extended pass. The implementation lives in `StepTxnOrchestrator` (Section C) as the unified `handle_work_completion` entry point that every ULFM collective invokes after `.wait()`.

D.3 Gradient restoration

Algorithm 5 enforces the correctness invariant of Section 4.2: every contribution admitted into the iteration sum must come from the same membership of PG_{cross}, or the per-microbatch contributions carry mismatched weights and the iteration gradient is corrupted. The world-epoch tag makes the

Algorithm 4 HANDLE_WORK_FAILURE(W).

Require: `Work` object W with the failure record produced by Record (Algorithm 2).
Build a `FailureEvent` e with $\langle W.\texttt{role_counts}, W.\texttt{contrib}, W.\texttt{at_boundary}, m, \epsilon_{\text{cur}}\rangle$.
$d \leftarrow$ POLICY_ADJUSTMENT(e).
if $e.\texttt{at_boundary}$ **then**
    install d's boundary-minor split on PG_{cross} so each rank queries the correct workload on the subsequent policy boundary step.
    quiesce PG_{cross} {stale buckets will be rolled back}
end if
latch the iteration's restore mode $\leftarrow d.\texttt{restore_mode}$.
invalidate the per-epoch "already reduced" bookkeeping (the new epoch makes it stale).

check local — a bucket is stale iff its tag predates the current epoch — and a stale bucket must be rewound from $S(b)$ before its content can be admitted under the repaired membership.

The mode latched by POLICY_ADJUSTMENT controls *when* the rewind fires. A non-boundary failure with a spare available leaves the iteration's total microbatch count unchanged, so the rewind plus a re-issued reduction must finish before the optimizer step — this is the blocking branch, which calls ULFM_ALLREDUCE synchronously on each stale bucket. A boundary failure already requires extra microbatches, so the rewind is scheduled on a side CUDA stream and overlapped with the extended pass's forward; the re-reduction then happens implicitly when the extended pass's all-reduce on the bucket fires on the repaired membership, with no extra round-trip needed. The implementation (Section C) is the orchestrator's `restore_gradients_blocking` and `restore_gradients_non_blocking` routines, dispatched on the latched `restore_mode`.

Algorithm 5 GRADIENT_RESTORATION.

Require: Bucket bookkeeping $\mathcal{B}$, current epoch ϵ_{cur}, latched restore mode.
if restore mode = SKIP **then return**.
$stale \leftarrow \{b \in \mathcal{B} : \epsilon(b) < \epsilon_{\text{cur}}\}$.
if restore mode = NON-BLOCKING **then**
    {policy boundary: overlap rewind with the upcoming extended-pass forward.}
    on a dedicated CUDA stream, copy $S(b) \to b$ for every stale b; record an event for the next backward to wait on.
    clear $\mathcal{B}$ and unquiesce PG_{cross} {the extended pass will repopulate $\mathcal{B}$ with fresh snapshots and re-reduce on ϵ_{cur} as it goes}
else
    {non-boundary: rewind *and* re-reduce before the optimizer step.}
    for all stale bucket b **do**
        copy $S(b) \to b$; then $W \leftarrow$ ULFM_ALLREDUCE(b, PG_{cross}) and wait.
        if W failed **then return** {the caller cross-syncs replicas and re-enters HANDLE_WORK_FAILURE }
    end for
    unquiesce PG_{cross}; restore mode $\leftarrow$ SKIP.
end if

D.4 Policy hooks

Algorithm 6 is the policy's response to a failure inside an iteration, and is the only place where the loop bound $P(\text{major})$ in Algorithm 1 can change. The two branches follow directly from whether a spare absorbed the loss. If yes (the non-boundary case), the failure is invisible to the iteration's totals: Record already promoted a spare in Algorithm 2, and from this point on it carries the vacated rank's microbatch quota. The policy returns with P unchanged — in particular $P(\text{major})$ unchanged — so when control returns to Algorithm 1 the outer **while** terminates as planned and the iteration commits with $\sum_r C_r = B$ unchanged. If no (the boundary case), the iteration must run extra microbatches to recover the lost contributions: the policy picks the smallest G_{ext} with $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} \geq B$, marks $W_{\text{cur}} \cdot G_{\text{ext}} - (B - C_{\text{cur}})$ survivors as *boundary minors* that contribute one fewer extra microbatch (so the total lands at exactly B), and grows $P(\text{major})$ by G_{ext} (boundary minors by $G_{\text{ext}} - 1$). When the inner bucket loop exits and the outer **while** re-tests $m < P(\text{major})$, the test is now true and the iteration runs the extended pass. The implementation (Section C) is `StaticWorldPolicy.on_failure`, with the boundary branch in `_on_policy_boundary`.

Algorithm 6 POLICY_ADJUSTMENT(e): in-iteration response (may grow $P(\text{major})$).

Require: A `FailureEvent` e produced by HANDLE_WORK_FAILURE.
Ensure: Updated P, ρ, plus a `PolicyDecision` carrying `restore_mode`.
1: **if not** e.`at_boundary` **then**
2: {a spare in the failed role was already promoted in Record; $P(\text{major})$ stays the same.}
3: refresh the policy's tracked counts (W_{cur}, spares) from e.`role_counts`.
4: **return** P, ρ unchanged (except the promoted spare); `restore_mode` = BLOCKING.
5: **else**
6: {spares of the failed role are exhausted; *grow* $P(\text{major})$ to extend the iteration.}
7: $G_{\text{ext}} \leftarrow$ smallest integer ≥ 1 with $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} \geq B$.
8: $n_{\text{bdry}} \leftarrow W_{\text{cur}} \cdot G_{\text{ext}} - (B - C_{\text{cur}})$ {boundary-minor count; 0 if B divides exactly}
9: $P(\text{major}) \leftarrow P(\text{major}) + G_{\text{ext}}$; $P(\text{boundary-minor}) \leftarrow P(\text{major}) - 1$; designate n_{bdry} survivors as boundary minors and refresh ρ on each rank.
10: **return** updated P, ρ; `restore_mode` = NON-BLOCKING.
11: **end if**

Algorithm 7 fires only once per policy-boundary iteration, after the optimizer step has committed: it installs the steady-state role layout the surviving world will run from the *next* iteration onwards. The choice of G_{cur} is the smallest factor that lets W_{cur} survivors cover B with a single integer grad-accum; the residue $R_{\text{cur}} = B - n_{\text{maj}} \cdot G_{\text{cur}}$ is absorbed by at most one minor. Any extra survivors are reserved as spares (mostly major-spares, with at least one minor-spare reserved when a minor exists) so the next failure has a chance of landing in the non-boundary branch of POLICY_ADJUSTMENT. The implementation (Section C) is `StaticWorldPolicy.advance_policy`, which the orchestrator drives from `after_successful_commit` and pushes into the C++ backend via `set_major_minor_split_with_spares`.

Algorithm 7 POLICY_ADVANCEMENT: post-boundary steady-state.

Require: Survivor count W_{cur}, target B.
Ensure: New role layout $(G_{\text{cur}}, n_{\text{maj}}, n_{\text{min}}, R_{\text{cur}}, n_{\text{ms}}, n_{\text{mi}})$ pushed to PG_{cross}.
1: $G_{\text{cur}} \leftarrow$ smallest integer with $W_{\text{cur}} \cdot G_{\text{cur}} \geq B$.
2: $n_{\text{maj}} \leftarrow \lfloor B/G_{\text{cur}} \rfloor$; $R_{\text{cur}} \leftarrow B - n_{\text{maj}} \cdot G_{\text{cur}}$; $n_{\text{min}} \leftarrow 1$ if $R_{\text{cur}} > 0$ else 0.
3: Allocate the remaining $W_{\text{cur}} - n_{\text{maj}} - n_{\text{min}}$ ranks as major-spares and minor-spares; reserve one minor-spare when $n_{\text{min}} = 1$ and $W_{\text{cur}} - n_{\text{maj}} - n_{\text{min}} \geq 2$, otherwise all major-spares.
4: Push the new layout to PG_{cross}, clear the policy-boundary flag, and update each surviving rank's role ρ accordingly.
5: **return** the new policy P and refreshed role ρ for this rank.

D.5 ADAPTIVEWORLDPOLICY: a strawman repair-and-continue baseline

Algorithm 8 is a drop-in replacement for Algorithm 6 that does the bare minimum: on any failure it returns BLOCKING restore with `at_boundary = false`, regardless of whether spares exist; POLICY_ADVANCEMENT is never called because no boundary is ever crossed. The iteration commits with $W_{\text{cur}} \cdot G_{\text{cur}} < B$ — the global batch shrinks with the world and the trajectory drifts. The implementation (Section C) is `AdaptiveWorldPolicy` on the same orchestrator and ULFM backend as the static policy. We keep it as an isolation baseline: pairing the same ULFM-guarded collective and in-step rewind with this minimal policy reproduces what prior keep-alive frameworks [19, 30] provide, and shows what versatile workload contributes on top.

Algorithm 8 ADAPTIVEWORLDPOLICY (strawman).

Require: A `FailureEvent` e.
1: **return** `PolicyDecision`(restore_mode = BLOCKING, at_boundary = **false**). {PG_{cross} was repaired in phase 2 of Algorithm 2; the iteration commits with effective batch $W_{\text{cur}} \cdot G_{\text{cur}} < B$.}

E Numerical Walk-through of a Boundary Step

This appendix traces the $W_{\text{init}} = 32$, $G_{\text{init}} = 8$ example of Figure 5 (so $B = 256$), making explicit how the contribution count $C_r(t)$ on each replica accumulates and how the world-epoch tagging

keeps every contribution under the same membership of PG_{cross}. Recall the iteration schedule from Algorithm 1: every replica runs all $P(\text{major})$ microbatches with local accumulation, and only at the last microbatch does the bucket loop fire per-bucket ULFM_ALLREDUCE on PG_{cross}, followed by a single barrier on PG_{intra} and a single ULFM_CONSENSUS on PG_{cross}.

Let $\mathcal{R} = \{r_1, \ldots, r_{32}\}$ denote the initial replicas. Let ϵ_0 be the world epoch on PG_{cross} before the failure, and write $\epsilon_1 = \epsilon_0 + 1$ for the post-repair epoch that the Repair phase of ULFM_ALLREDUCE installs. Suppose that at iteration $t^\star$ replica r_{32} fails partway through the bucket loop — specifically, during the all-reduce of some bucket $b^\star$.

Pre-failure iterations. For every iteration $t < t^\star$, every replica is a major with $P(\text{major}) = G_{\text{init}} = 8$, so $C_r(t) = 8$ for all r. The bucket loop reduces every bucket cleanly under epoch ϵ_0 and the iteration commits with $\sum_{r=1}^{32} C_r(t) = 32 \cdot 8 = 256 = B$, satisfying Equation (1).

State at the moment of failure (iteration $t^\star$). By the time the bucket loop is running, every replica has already completed all 8 microbatches of forward+backward and has each microbatch's contribution recorded against its `contrib` counter on PG_{cross}. The Detect probe of ULFM_ALLREDUCE on $b^\star$ surfaces r_{32}'s failure; Repair shrinks PG_{cross} to the 31 survivors and bumps the world epoch to $\epsilon_1 = \epsilon_0 + 1$; Record runs the role-flag all-reduce among survivors and reports `contrib` $= C_{\text{cur}} = 31 \cdot 8 = 248$ (the contributions r_{32} had logged are dropped because r_{32} is no longer in the membership over which the census all-reduce runs). `at_boundary` is set to **true**: r_{32} was a major and no major-spare exists at iteration $t^\star$ to absorb the loss. At this moment, the survivors hold buckets in three positions relative to $b^\star$:

- *Buckets that completed before $b^\star$ (epoch $\epsilon_0 < \epsilon_1$).* These reduced cleanly under the full 32-replica membership and would now carry the wrong weight if mixed with 31-replica reductions in the same sum. They are stale per Algorithm 5: their pre-reduce snapshots $S(b)$ are scheduled to be copied back into the bucket buffers and will be re-reduced at ϵ_1.
- *Bucket $b^\star$ itself.* Its all-reduce returned the failure signal, so its post-reduce content is undefined; it is rewound from $S(b^\star)$ and will be re-reduced at ϵ_1.
- *Buckets that hadn't been visited yet.* Their snapshots were never taken, and their contents are still locally accumulated partial sums; they will get fresh snapshots tagged ϵ_1 when the bucket loop resumes after recovery.

The boundary extension. POLICY_ADJUSTMENT receives $C_{\text{cur}} = 248$, $W_{\text{cur}} = 31$ and `at_boundary = true`, and chooses the smallest G_{ext} with $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} \geq B$:

$$G_{\text{ext}} = \left\lceil \frac{B - C_{\text{cur}}}{W_{\text{cur}}} \right\rceil = \left\lceil \frac{256 - 248}{31} \right\rceil = 1.$$

The overshoot is $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} - B = 248 + 31 - 256 = 23$, so 23 of the 31 survivors are reassigned from major to *boundary minor* — the role that contributes $G_{\text{ext}} - 1 = 0$ extra microbatches. The remaining 8 survivors stay majors and contribute the full $G_{\text{ext}} = 1$ extra microbatch each. Equivalently, $P(\text{major})$ grows from 8 to 9 while $P(\text{boundary-minor})$ stays at 8. GRADIENT_RESTORATION is latched in non-blocking mode and schedules the rewinds on a side CUDA stream; the bucket bookkeeping $\mathcal{B}$ is cleared.

The extended pass. Control returns to Algorithm 1's outer **while**, which now re-tests $m < P(\text{major})$ against the new $P(\text{major}) = 9$ and re-enters with $m = 9$. Every survivor runs forward and backward on microbatch 9. The 8 majors locally accumulate (their $P(\rho) = 9$); the 23 boundary minors zero this microbatch's gradient (their $P(\rho) = 8$). After backward, the bucket loop fires again on PG_{cross} at epoch ϵ_1, takes fresh snapshots for every bucket, and reduces each one cleanly — there is no extra round of re-reduction needed, because the rewinds of stale buckets have already overlapped with this microbatch's forward and the buckets now carry the union of all 9 microbatches' (for majors, boundary-minors still have 8) worth of locally-accumulated contributions. The post-loop barrier on PG_{intra} and ULFM_CONSENSUS on PG_{cross} pass cleanly, and the iteration commits its optimizer step.

Verifying the invariant at iteration $t^\star$. Every bucket admitted into iteration $t^\star$'s gradient was reduced exactly once under the 31-replica membership at epoch ϵ_1. Counting per-replica contributions,

$$\sum_{r \in \text{survivors}(t^\star)} C_r(t^\star) = \underbrace{31 \cdot 8}_{\substack{31 \text{ survivors @ } 8}} + \underbrace{8 \cdot 1}_{\substack{8 \text{ majors} \\ \times 1 \text{ extra}}} + \underbrace{23 \cdot 0}_{\substack{23 \text{ bdry-min} \\ \times 0 \text{ extra}}} = 248 + 8 + 0 = 256 = B,$$

as Equation (1) demands. The per-replica counts — 8 replicas at 9 microbatches and 23 replicas at 8 microbatches — match panel (ii) of Figure 5 exactly. r_{32}'s data partition drops out of the run for good from iteration $t^\star + 1$ onwards, while the 31 survivors advance through their own partitions slightly faster than the failure-free schedule prescribed. Because the pre-training stream is effectively infinite, this is statistically indistinguishable from having sampled a different random shuffle of the same stream from the outset; Section F formalizes the argument.

Restore-stream timing. The stale-bucket rewinds run on a CUDA stream disjoint from the compute stream, kicked off as soon as GRADIENT_RESTORATION latches the non-blocking mode. The rewind itself is a snapshot-to-buffer memcpy at typical bucket sizes ($O(100\,\text{MB})$ of fp32). The compute stream reaches the first backward on the extra microbatches only after the rewind stream has finished.

Post-boundary steady state, iteration $t^\star+1$. Once iteration $t^\star$ commits, POLICY_ADVANCEMENT is invoked with $W_{\text{cur}} = 31$, $B = 256$ and previous $G_{\text{cur}} = 8$. It bumps G_{cur} until $W_{\text{cur}} \cdot G_{\text{cur}} \geq B$, giving $G_{\text{cur}} = 9$. Then $n_{\text{maj}} = \lfloor 256/9 \rfloor = 28$ and $R_{\text{cur}} = 256 - 28 \cdot 9 = 4$, so $n_{\min} = 1$. The remaining $W_{\text{cur}} - (n_{\text{maj}} + n_{\min}) = 2$ ranks are spares — 1 major-spare and 1 minor-spare, since $n_{\min} = 1$ and $W_{\text{cur}} - (n_{\text{maj}} + n_{\min}) > 1$. This new layout is pushed to PG_{cross} and matches panel (iii) of Figure 5. Verifying the invariant for the new steady state,

$$\sum_{r \in \text{survivors}(t^\star+1)} C_r(t^\star+1) = \underbrace{28 \cdot 9}_{\text{majors}} + \underbrace{1 \cdot 4}_{\text{minor at } R_{\text{cur}}} + \underbrace{1 \cdot 0 + 1 \cdot 0}_{\text{spares}} = 252 + 4 + 0 = 256 = B.$$

The two spares run their full forward and backward but have their gradient buffer zeroed at ULFM_ALLREDUCE time (because `is_spare` is true and `at_boundary` is false in the steady state); they are immediately ready to absorb the next failure, since promotion in Record simply clears `is_spare` and the spare's already-computed gradients were snapshotted thus can be rewound to become its contribution. The next failure that lands in a role that has a spare available therefore stays in the non-boundary branch of POLICY_ADJUSTMENT, and the iteration commits without extending $P(\text{major})$ at all — exactly the *non-boundary-crossing failure* sketched in panel (iv) of Figure 5.

F Proof Sketch of the Stochastic-Equivalence Guarantee

We make the stochastic-equivalence property of Section 4 precise. The claim is *not* that REcoVER reproduces a failure-free reference run step-for-step or bitwise; rather, that as a distribution over the same data stream, every iteration of a REcoVER run draws its gradient from the same distribution as the corresponding reference iteration. Throughout, we re-use the notation of Section 4.3: $W_{\text{init}}, G_{\text{init}}, B = W_{\text{init}} \cdot G_{\text{init}}$ are the initial replica count, the initial grad-accum factor, and the target global batch; $W_{\text{cur}}(t), G_{\text{cur}}(t)$ are the current values at iteration t; $C_r(t)$ is the contribution count of replica r at iteration t (its role-assigned microbatch count for a major or minor, zero for a spare); and the cross-replica process group PG_{cross} has world epoch $\epsilon_{\text{cur}}(t)$, incremented on every successful repair.

Setup. Fix a deterministic model with parameters θ_t at iteration t and per-example gradient $g(\theta_t; x)$, and fix a deterministic pre-training data stream $X = (x_0, x_1, \dots)$ partitioned across the initial replicas, with each replica owning a disjoint slice of X. Let $\mathcal{R}(t)$ denote the set of replicas alive at iteration t, and let $\phi_t : \mathcal{R}(t) \to \mathcal{P}(X)$ be the explicit replica-to-microbatch assignment whose gradient contributes to iteration t's gradient sum assignment, with $|\phi_t(r)| = C_r(t)$. For a major or a minor, $\phi_t(r)$ is the set of microbatches r runs forward+backward on; for a major-spare or minor-spare, $\phi_t(r) = \emptyset$, since the spare's gradient is zeroed at all-reduce time. Define

$$\mathcal{A}_{\text{REcoVER}}(t) = \bigsqcup_{r \in \mathcal{R}(t)} \phi_t(r), \qquad \mathcal{A}_{\text{ref}}(t) = \text{the analogous multiset for a failure-free reference run,}$$

where $\bigsqcup$ is the disjoint union of the per-replica assignments. Both are multisets of microbatches drawn from X.

Claim. Let Σ be any failure schedule that retains at least one replica at every t. Then for every iteration t, $|\mathcal{A}_{\text{ReCoVer}}(t)| = B = |\mathcal{A}_{\text{ref}}(t)|$ and both multisets are subsets of X without repetition. The multisets differ only in which slice of X each microbatch is drawn from: failed replicas' slices drop out of X entirely once the replica is lost, and survivors advance through their own slices slightly faster than the reference would prescribe. Under the standard exchangeability assumption on the pre-training data stream, $\mathcal{A}_{\text{ReCoVer}}(t)$ and $\mathcal{A}_{\text{ref}}(t)$ have the same distribution over X for every t, so the iteration gradient $\frac{1}{B}\sum_{x\in\mathcal{A}_{\text{ReCoVer}}(t)} g(\theta_t; x)$ is drawn from the same distribution as the reference's $\frac{1}{B}\sum_{x\in\mathcal{A}_{\text{ref}}(t)} g(\theta_t; x)$.

Proof sketch. The protocol of Section 4 maintains, for every iteration t, the explicit replica-to-microbatch assignment $\phi_t : \mathcal{R}(t) \to \mathcal{P}(X)$ with $|\phi_t(r)| = C_r(t)$ and $\bigsqcup_r \phi_t(r) = \mathcal{A}_{\text{ReCoVer}}(t)$. We verify $|\mathcal{A}_{\text{ReCoVer}}(t)| = B$ at every iteration by cases.

Case (a): no failure at t. The role assignment is inherited from iteration $t-1$. Majors and (the at most one) minor contribute their role-assigned microbatch counts; spares contribute $C_r(t) = 0$ but still execute forward and backward, with their gradient buffer zeroed at ULFM_ALLREDUCE time. The world epoch is unchanged across the iteration, so every bucket is reduced exactly once at the same epoch $\epsilon_{\text{cur}}(t)$, and $\sum_r C_r(t) = n_{\text{maj}} \cdot G_{\text{cur}} + n_{\text{min}} \cdot R_{\text{cur}} = B$ holds by construction of the policy steady state.

Case (b): failure at t, spare available in the failed role. The C++ helper `record_and_handling_failure` (Section C) repairs PG_{cross}, increments the world epoch from ϵ_{cur} to $\epsilon_{\text{cur}} + 1$, and atomically promotes a spare into the vacated role, all within the same allreduce path. The promoted spare had been zeroing its contributions only at ULFM_ALLREDUCE time, so its gradient bytes already exist locally; rewinding from snapshots simply admits them. Every bucket whose snapshot was tagged with the pre-failure epoch is rewound from its snapshot and re-reduced under $\epsilon_{\text{cur}} + 1$ via Algorithm 5, so every bucket in the iteration is summed under the same epoch. Because the spare's contribution count exactly replaces the failed replica's, $\sum_{r\in\mathcal{R}(t)} C_r(t) = B$.

Case (c): failure at t, spares of the failed role are exhausted (policy boundary). The policy detects the boundary via phase 3 of ULFM_ALLREDUCE, picks the smallest integer $G_{\text{ext}} \geq 1$ with $C_{\text{cur}} + W_{\text{cur}} \cdot G_{\text{ext}} \geq B$, and designates $W_{\text{cur}} \cdot G_{\text{ext}} - (B - C_{\text{cur}})$ survivors as boundary minors that contribute $G_{\text{ext}} - 1$ extra microbatches; the other survivors contribute the full G_{ext}. By construction of the boundary-step counts,

$$\sum_{r\in\mathcal{R}(t)} C_r(t) \;=\; C_{\text{cur}} \;+\; (\text{majors}) \cdot G_{\text{ext}} \;+\; (\text{boundary minors}) \cdot (G_{\text{ext}} - 1) \;=\; B.$$

As in case (b), every bucket whose snapshot tag predates the post-repair epoch is rewound and re-reduced under $\epsilon_{\text{cur}} + 1$ via Algorithm 5, so every bucket admitted into iteration t is reduced under the same membership and none of the failed replica's pre-failure work is retained. The iteration gradient is therefore a uniformly-weighted sum of exactly B survivor microbatches.

The three cases together establish $|\mathcal{A}_{\text{ReCoVer}}(t)| = B$ for every t. By the disjointness of ϕ_t, no microbatch in X is admitted twice in the same iteration. Across the run, the multiset $\bigsqcup_t \mathcal{A}_{\text{ReCoVer}}(t)$ is therefore a subset of X that omits the failed replicas' slices entirely; because each replica owns a disjoint slice of X, dropping a replica removes only its slice and the survivors continue to consume their own slices in order. The pre-training stream is effectively infinite (the corpus is never lapped), so the remaining-survivors stream is statistically indistinguishable from a different random shuffle of X. Under exchangeability of X, $\mathcal{A}_{\text{ReCoVer}}(t)$ and $\mathcal{A}_{\text{ref}}(t)$ have the same distribution over X at every t, which is the distributional-equivalence claim. □

G Supplementary Evaluations

G.1 Evaluation under i.i.d. Failures

The failure schedule of our 512-GPU experiments is not fixed: each failure is placed at a randomly chosen step within its interval, which lets the run complete all desired failures within a cluster job's wall-time. As a reference, we present an earlier 128-GPU experiment in which each device fails independently at each step with a pre-defined failure probability. The sampled failure steps are 35, 103, 118, 136, 139, 195, 252, 346 and 370; only 3 of the 9 failures fall after step 200, as fewer and

fewer GPUs are alive and the sampling often hits ranks of already-failed replicas, which makes the schedule much sparser. Here each step's global batch size is only 64, much smaller than the 8192 of our 512-GPU experiment, so each step's gradient has a higher variance and each step's loss value differs slightly more. Under this i.i.d. failure schedule, RECOVER still produces a training curve statistically identical to the failure-free reference (Table 3).

Step	50	100	150	200	250	300	350
Failure-free reference	7.66	6.69	5.98	5.50	5.31	4.98	4.75
RECOVER	7.65	6.75	6.02	5.51	5.24	4.94	4.74

Table 3: Training loss under the i.i.d. failure schedule of Section G.1 (128 GPUs).

G.2 Cost of In-Step Recovery

Gradient snapshotting and restoration incur no additional communication, as each rank locally copies its own pre-all-reduce gradient shard. Their peak GPU-memory overhead is one local gradient copy, and their runtime is the HBM traffic divided by the memory bandwidth. For our 7B model with TP$= 4$, PP$= 2$, balanced partitioning and FP32 gradient accumulation, the snapshot takes $7\text{B}/(4 \times 2) \times 4$ bytes $= 3.5\,\text{GB}$ per GPU; taking it reads and writes $3.5\,\text{GB}$ each, i.e., about $2 \times 3.5/1555 \approx 4.5\,\text{ms}$ on an A100, and the same holds for restoration. Both happen once per iteration, which lasts tens to hundreds of seconds, so their cost is negligible.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: See methodology in Section 4 and evaluations in Section 5 and Section A.3.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: See Section 6, which discusses the dependence on ULFM, the replica as the unit of recovery, and why removed replicas do not rejoin.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: See Section F.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: See Section 5 and Section A.1 for full details of our evaluations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The code is available at `https://github.com/alvin-zyl/ReCoVer`, with instructions to install and run it. The C4 dataset we use is public.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: See Section 5 and Section A.1 for full details of our evaluations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: The statistical significance of our measured system performance gain is obvious in common sense. It is not like metrics such as accuracy that varies across different random seeds, it's the actual training throughput that reflects how fast the system makes progress, which is regarded as deterministic in common sense. Moreover the performance gap is $> 2\times$, same goes with the 74.9% more processed tokens on order of hundreds of millions of tokens. We believe these results are more than qualified to be deemed significant.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: See Section 5 and Section A.1 for full details of our evaluations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We certify this research was conducted in compliance with Neurips code of ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Our method is to reduce the massive computing resource wasted by contemporary systems that do not tolerate failures. See a general discussion in Section 1.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: Our research does not release such model or data.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We use open-source code/data, which have been properly credited, mentioned, and respected.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release the code of RECOVER under the Apache License 2.0. Its README documents the installation, the usage and the container.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: We perform no such experiment.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: Our research does not involve such subject.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: We only used LLM in standard ways, and LLM is not involved in our core methodology.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.