ElasWave: An Elastic-Native System for Scalable Hybrid-Parallel Training

Xueze Kang[1†], Guangyu Xiang[1†], Yuxin Wang[4*], Hao Zhang[4], Yuchu Fang[4], Yuhang Zhou[3], Zhenheng Tang[2], Youhui Lv[5], Eliran Maman[7], Mark Wasserman[7], Alon Zameret[7], Zhipeng Bian[6], Shushu Chen[5], Zhiyou Yu[5], Jin Wang[5], Xiaoyu Wu[4], Yang Zheng[4], Chen Tian[3], Xiaowen Chu[1*]

[1]*HKUST(GZ)* [2]*HKUST* [3]*NJU*

Huawei [4]*TTE Lab* [5]*ICT BG* [6]*Cloud* [7]*TRC Team*

Abstract

Large-scale LLM pretraining today spans 10^5–10^6 accelerators, making failures commonplace and elasticity no longer optional. We posit that a elastic-native training system must simultaneously ensure (i) Parameter Consistency, (ii) low Mean Time to Recovery (MTTR), (iii) high post-change Throughput, and (iv) Computation Consistency. This objective set not has never been jointly attained by prior work. To achieve these goals, we present ElasWave, which provides per-step fault tolerance via multi-dimensional scheduling across Graph, Dataflow, Frequency, and Random Number Generation. ElasWave resizes and reshards micro-batch workloads while preserving the global batch size and gradient scale; it performs online pipeline resharding with asynchronous parameter migration, interleaving ZeRO partitions so recovery reduces to disjoint rank-to-rank transfers. It further uses DVFS to absorb pipeline bubbles and reshards RNG to keep consistent computations. A dynamic communicator enables in-place communication group edits, while per-step in-memory snapshots support online verification and redistribution. We evaluated ElasWave on 96 NPUs and benchmarked against state-of-the-art baselines: throughput improves by 1.35× over ReCycle and 1.60× over TorchFT; communicator recovery completes within one second (up to 82 × /3.6× faster than full/partial rebuilds); migration MTTR drops by as much as 51%; and convergence deviation is reduced by approximately 78%.

Keywords: Hybrid Parallelism, Elastic Training, Large Language Models, Fault Tolerance.

1 Introduction

LLM training [7, 32, 37, 56, 65] has progressed from tens of thousands to 10^5 accelerators, with roadmaps aiming at 10^6 (e.g., the Stargate system) [44]. At current scale, Google Cloud grows a single 8,960-chip TPU pod to tens of thousands via Multislice [10], while Huawei's UnifiedBus-based Atlas SuperPoDs (8,192/15,488 NPUs) aggregate into SuperClusters at 5×10^5 to 10^6 scales [18]. With hyperscale clusters and preemptible clouds, month-long pretraining is routine where fail-stop [16, 22, 24, 58, 62] and fail-slow [31, 63, 64] events are commonplace. For instance, a 16,384-H100 Llama-3 run reported interruptions roughly every three hours, with nearly half due to GPU or HBM issues [24]. Elastic training is therefore a first-class requirement: Systems must maintain progress despite resource variation, reconfigure [9, 21, 57], sustain throughput [9], and preserve convergence [29, 48].

We define four objectives for an elastic-native training system in production: (i) *Parameter Consistency* across hybrid parallelism, (ii) low *Mean Time To Recovery (MTTR)*, (iii) high *Throughput* after scale-in/scale-out, and (iv) *Computation Consistency* that preserves the optimization trajectory of a static run [29]. These objectives are coupled, and no prior system achieves all four simultaneously. Oobleck [21] ensures parameter consistency via redundancy but imposes steady-state overhead. Other works [29][9] preserve DP determinism, whereas do not guarantee consistency for sharded state [58]. The academic state-of-the-art (SOTA), ReCycle [9] avoids restart by using pipeline bubbles for intra-stage rerouting. However, when training scales, the tiny bubble budget is quickly exhausted, leading to stragglers and out-of-memory (OOMs). The industrial SOTA, TorchFT [51], sustains step-level (i.e., one training iteration) progress by dropping and rejoining DP replicas, which wastes substantial compute resources and creates pronounced throughput cliffs [51].

This paper presents ElasWave as a comprehensive solution. It introduces *multi-dimensional scheduling* that delivers per-step fault tolerance by coordinating four axes—dataflow, graph, accelelator frequency, and RNG. (i) *Dataflow*: we resize and reshard micro batches while preserving global batch size and gradient scale for immediate continuation. (ii) *Graph*: we propose online pipeline resharding with layer migration to restore load balance. We introduce non-blocking migration to overlap with compute, and interleave ZeRO state sharding to reduce recovery to disjoint rank-to-rank sends, avoiding intra-group reshaping and large transfers. (iii) *Hardware*: frequency scaling removes residual bubbles after parameter recovery. (iv) *RNG*: RNG resharding maintains step-identical random streams after membership changes and bounds numerical drift. To further improve recovery efficiency, we also introduce a dynamic communicator that reuses existing connections to avoid full rebuilds. Together with per-step in-memory snapshots and live remapping, parameters are

†Equal contribution.

*Corresponding authors: yuxin.wang11@huawei.com, xwchu@hkust-gz.edu.cn.

Table 1. Capability matrix of elastic-training systems. Columns summarize support for: *MTTR*, *Throughput*, *Computation Consistency*, and *Parameter Consistency*. Symbols: ✓ supported, × unsupported, △partial. ElasWave is the only system covering DP&PP granularity with joint data/graph/hardware planning, RNG consistency, and both rollback and live resharding.

	MTTR		Throughput				Computation Consistency		Parameter Consistency	
System	**Online**	**Optim.**	**Granularity**	**Data**	**Graph**	**Hardware**	**Systematic**	**Numerical**	**Per-step Rollback**	**Live Reshard**
TorchElastic [46]			DP						×	
DLRover [61]			DP						△	
ByteCheckpoint [58]	×		×		×		×		✓	
EasyScale [29]		×	DP	×			✓			
Oobleck [21]	✓		DP&PP		✓		×			
ReCycle [9]	✓	△	DP&PP	✓		×	△	×	×	×
TorchFT [51]	✓	×	DP	×	×		×			
ElasWave	✓	✓	**DP&PP**	✓	✓	✓	△	✓	✓	✓

verified, redistributed, and loaded on-the-fly to achieve efficient elasticity. As shown in Table 1, ElasWave is, to our knowledge, the first scalable, elastic-native training system on XPU clusters.

ElasWave is built with two components: *ElasWave Agent* and *ElasWave Core*. The Agent detects interruptions ; the Core plans and executes elastic responses:

(i) Scheduling: ElasWave Core performs elastic multi-dimensional scheduling within a single step, jointly deciding *Dataflow, Graph, DVFS, RNG* and emitting an *executable recovery plan* that optimizes parameter consistency, low MTTR, post-change throughput, and computation consistency.

(ii) Executions: ElasWave Core minimizes MTTR by (i) maintaining step-level in-memory snapshots with live remap, (ii) using a *Dynamic Communicator* to edit groups in place for sub-second recovery, and (iii) *overlapping* interleaved stage resharding to avoid blocking training.

(i) Implementations: ElasWave provides end-to-end deployment on an Ascend-910B NPU cluster, with hierarchical interfaces designed on CANN [19], Torch-npu [45], Megatron [40] with 20k+ loc.

(iv) Results: We evaluate ElasWave in LLM training with SOTA hybrid parallel setups [49, 55] across production traces. On our testbed, the throughput gains outperform ReCycle [9] by 1.35× and TorchFT [51] by 1.60×, while preserving parameter consistency. For MTTR, communicator recovery completes in sub-second time, improved by up to 82× and 3.6× compared with full and partial rebuilds method. Non-blocking migration with interleaved ZeRO cuts layer migration MTTR by up to 51% compared with blocked migration under vanilla ZeRO. RNG Resharding reduces convergence deviation by 78%, improving computation consistency.

2 Principles of An Elastic-Native System

Building an elastic-native system requires end-to-end consideration during the system design. The system must simultaneously deliver per-step parameter consistency, low MTTR, high throughput, and convergence consistency. Figure 1 is an example in ReCycle [9] that achieves fast failure recovery but introduces OOM and straggler issues with cummulative micro-batches in the cool-down phase in production.

Efficient Recovery with Consistent Parameters. Reconfiguring hybrid-parallel training (DP+PP+ZeRO/FSDP) *on the fly*, without a restart, is challenging: it requires reshaping PP by redistributing layers, splitting/merging DP groups to adjust replication, and reassigning sharded optimizer/gradient states. These operations are typically heavyweight, the recovery process of which includes large parameter migration and global coordination to overlap communication with compute. Prior systems either freeze the logical layout [29], precompute a limited set of templates [21], or handle only intra-stage DP failover given a surviving replica [9], leaving production training without end-to-end sharding/loading solutions.

Takeaway. An elastic-native system must generalize to arbitrary scale-in/out via precise state management and accelerated transfers *without* interrupting the next update. Recovery time should be itemized by component and minimized.

Computation Consistency. Elastic scaling can silently perturb both statistical and numerical consistency relative to a fixed-configuration run. To preserve trust, the effective

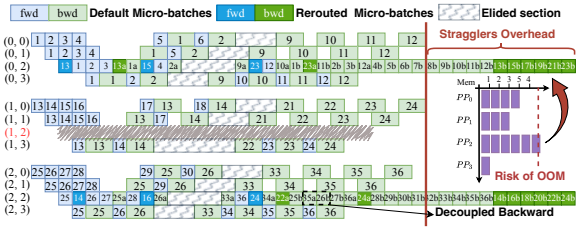


Figure 1. Pipeline schedule for ReCycle after a failure at node (1,2). ReCycle reroutes the failed rank's work to peers in the same stage (e.g., (0,2) and (2,2)), creating a straggler. While its decoupled backward pass creates bubbles to absorb the extra work, the large number of rerouted micro-batches quickly exhausts this bubble budget. The strategy also extends activation lifetimes, which increases memory pressure and risks Out-of-Memory (OOM) errors.

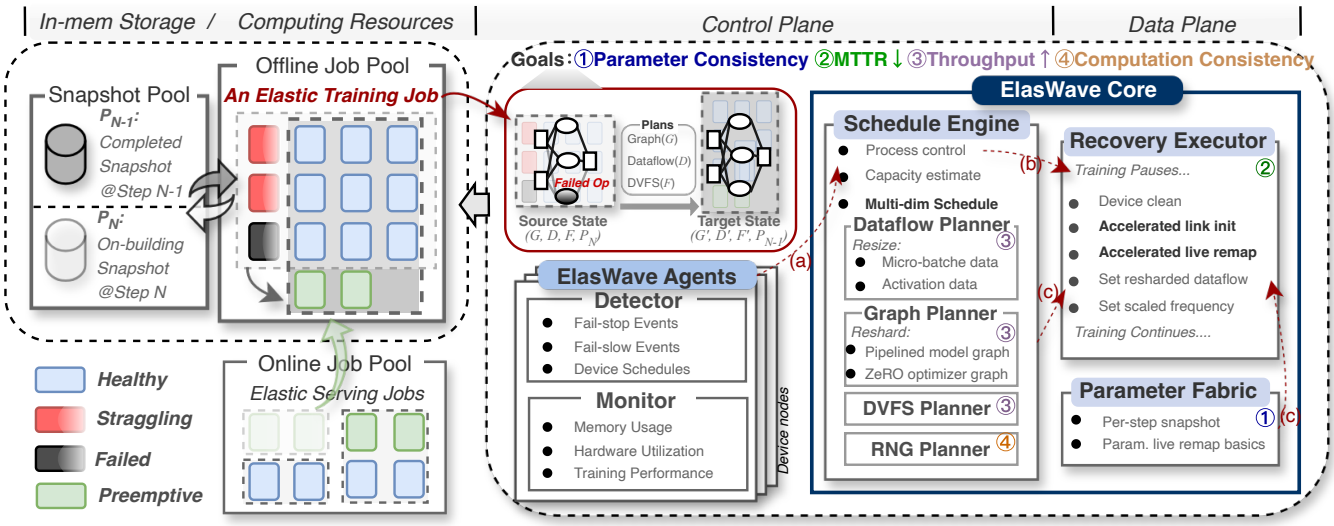


Figure 2. System architecture of ElasWave, illustrating the elastic recovery workflow and its triggers (fail-stop, fail-slow, and resource-scheduling signals). (a) When the ElasWave Agent detects a failure, straggler, or scheduling signals, it reports the current cluster state to the ElasWave Core. The Core then generates a multi-dim plan to optimize four key goals. (b) The Engine first pauses the training job via the Recovery Executor. (c) The recovery plan is dispatched to the Recovery Executor. The Executor uses the plan to perform an accelerated live remap, reconfigure links, and set the new dataflow, using state provided by the Parameter Fabric from the in-memory Snapshot Pool. Once the cluster is reconfigured, training resumes.

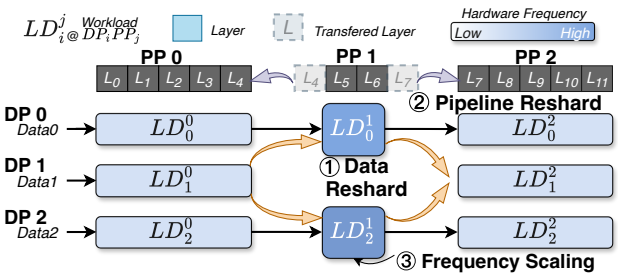


Figure 3. To optimize throughput, ElasWave's multi-dim scheduling combines three strategies. After a failure, it first performs *Data Reshard* in DP domain (①), then uses *Pipeline Reshard* in PP domain (②) to balance the workload, and finally eliminates remaining pipeline bubbles by *DVFS* (③).

batch size, RNG streams, execution order, and communication/reduction ordering must remain minimally perturbed; otherwise elasticity risks loss spikes or silent data corruptions [29]. We therefore treat *convergence invariance* as the first constraint in our elastic scheduler.

Takeaway. In ElasWave, we keep the computation graph and dataflow as regular as possible so elasticity remains traceable, and we implement deterministic hybrid-parallel RNG remapping aligned with each scheduling decision.

Maximize Throughput Under Constraints. Achieving high post-change throughput is a multi-objective scheduling problem: decisions in one dimension can conflict with others. Meeting all four goals therefore requires a careful systems design that coordinates these dimensions and integrates ideas from fault tolerance (fast recovery [13, 60], redundancy [57]), high-performance computing (load balancing [47, 67], scheduling [33, 69]), and ML engineering (statistical validity [48], reproducibility [52]) into a single framework.

Takeaway. Guided by multi-dimensional scheduling, our system jointly reasons about data, model, hardware, and RNG to navigate trade-offs that one-dimensional approaches cannot handle. ElasWave introduces online planners that, upon each resource change, compute a new hybrid-parallel plan (including device mappings) to maximize throughput under the current hardware pool, and seamlessly transition the running job to that plan.

The following sections will detail how our proposed system addresses each of these challenges, enabling on-the-fly elastic training for LLMs without sacrificing availability, efficiency, or model quality.

3 System Overview

ElasWave is an elastic-native LLM training system that delivers per-step fault tolerance via multi-dimensional scheduling across four axes: data, computation graph, device frequency, and RNG (Figure 2). It delivers per-step fault tolerance while jointly delivering (1)*parameter consistency* (across DP/PP/TP with ZeRO/FSDP states), (2)*low mean time to recovery (MTTR)*

at disturbances, (3)*high post-change throughput*, and (4)*computation consistency*—the resumed run follows the same optimization trajectory as fault-free training.

3.1 System Context

ElasWave operates on a shared resource pool whose capacity evolves under fail-stop, fail-slow, and scheduler-driven scale changes (including preemption), which we call elastic events. Training uses hybrid parallelism with ZeRO/FSDP sharding. A Snapshot Pool maintains per-step in-memory optimizer snapshots. Elastic events can interrupt or slow down the training process. We therefore require a system that absorbs these disturbances and resumes progress while simultaneously preserving the design goals.

3.2 System Components

ElasWave is an end-to-end system with a control plane and a data plane. ❶ **Control plane** detects elastic events with *Agent* and plans reconfiguration with *Schedule engine*;

Agent is an engineering-heavy runtime co-located with each worker. It continuously monitor failures and stragglers by hooking device/host/interconnect health probes and validating liveness, while collecting memory usage, hardware utilization, and step-level performance metrics. It also listen for scheduling/preemption/resizing signals and relay them to the Core.

Schedule engine governs the process control and produces a recovery plan after an elastic event. During planning, it evaluates the available device memory to ensure that the chosen actions do not exceed capacity constraints.

Dataflow planner decides the redirection of dataflow under elasticity: it adjusts micro batch routing across pipeline stages, with the implied redistribution of activations.

Graph planner assigns the computation graph: it repartitions the pipeline and maps partitions to hardware, and it determines the placement of ZeRO optimizer shards within each data-parallel group.

DVFS planner selects post-event frequency settings, computing the upclock values that best remove residual bubbles while respecting power and thermal limits.

RNG planner reshards the RNG state to remain consistent with dataflow and graph changes: it aligns per-sample/per-layer RNG usage so that randomness before and after elasticity is equivalent.

❷ **Data plane** executes recovery actions with *Recovery executor* and maintains redundancy with *Parameter fabric* for rapid resumption. It executes plans with minimized MTTR and consistent parameter versions.

Recovery executor execute the plan: it pauses training, sanitizes failed devices, restores communicators, reconstructs graph partitions, applies dataflow adjustments, migrates parameters, sets device frequencies, and then resumes training.

Parameter fabric maintains per-step snapshots of optimizer state and redundantly backs them up across nodes; upon shrink under ZeRO, it uses these snapshots to rebuild missing shards and reestablish optimizer integrity within each data-parallel group.

3.3 Workflow

As illustrated in Fig. 2, when the *Agents* detect a failure, straggler, or scheduling signal and report a resource-pool change, the Core's *Schedule Engine* synthesizes a multi-dimensional plan—redirect dataflows, repartition the pipeline and migrate layers, set DVFS to recover high throughput, and perform RNG Resharding for computation consistency—under capacity checks. At a step boundary the *Recovery Executor* briefly pauses the job, repairs connectivity by editing communicator links in place, uses the *Parameter Fabric* to live-remap missing ZeRO shards from the in-memory *Snapshot Pool* to ensure parameter consistency, and then applies the planned dataflow/graph/DVFS/RNG changes; communication recovery and non-blocking layer migration are optimized to minimize MTTR, after which training resumes.

4 Schedule Engine

Elastic events change the available resource, invalidating the current training configuration. The schedule engine schedules a new training configuration based on a new resource pool along four dimensions—Dataflow, Graph, DVFS, and RNG. This coordinated plan restores a runnable setup while meeting the four targets: parameter consistency, low MTTR, high post-change throughput, and computation consistency.

4.1 Dataflow Planner

Upon a node failure, its micro batch dataflow (and associated activations) must be promptly rerouted to surviving nodes to sustain training progress. ElasWave performs *micro batch resizing* instead of micro batch number rerouting in ReCycle for multi-dimensional scheduling that avoids stragglers and OOM issues. The micro batch previously handled by the failed rank is sliced along the batch dimension evenly into DP portions and added to the surviving ranks' micro batch sizes (Fig. 3①). For example, with DP = 3 and per-rank micro batch size = 2, a single failure yields DP = 2 and size = 3; the product DP × micro batch size remains constant (6), preserving the effective global batch and gradient scale. Resizing leaves the pipeline topology and choreography unchanged and only adjusts each micro batch's service time. Activations are produced and released by the standard forward/backward schedule, avoiding problems such as tail accumulation and extra peak memory that occur in Recycle (Figure 4(i)).

4.2 Graph Planner

The micro batch resizing policy from Dataflow Planner allows recovering at low MTTR, but applying it in isolation introduces new performance and memory hazards. When

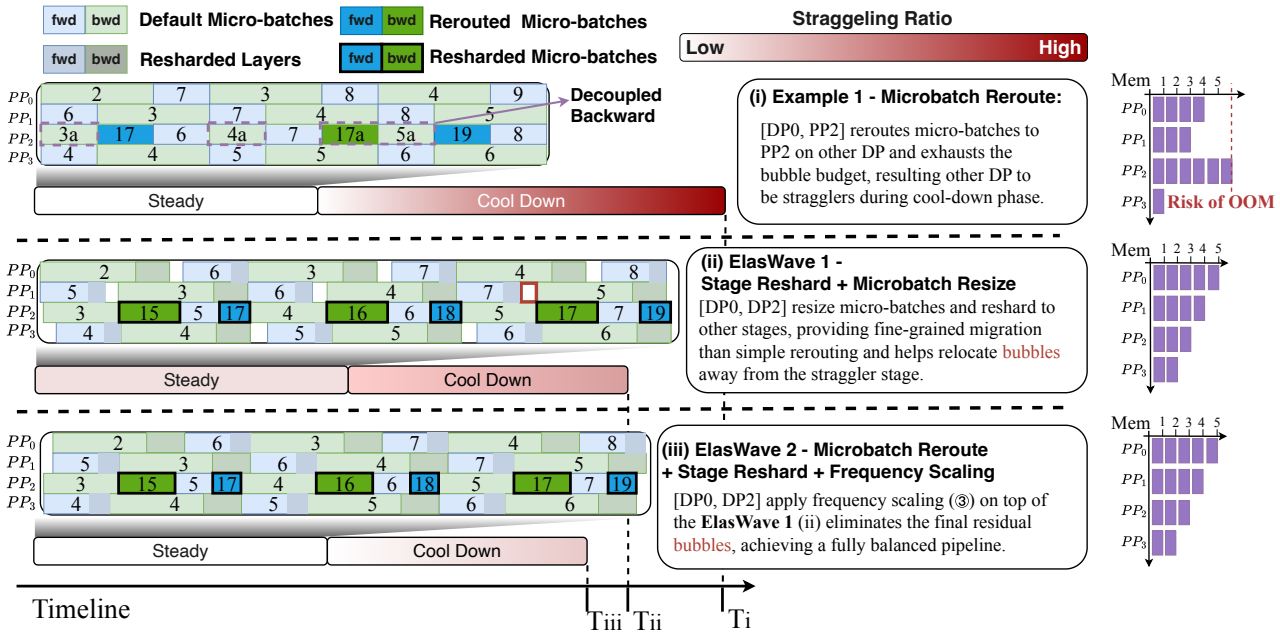


Figure 4. Comparison of pipeline schedule examples. The steady phase illustrates how multi-dim scheduling in (ii)(iii) avoids the stragglers and OOM issues in data rerouting in (i), achieving efficient and reliable training states step-by-step. The total execution times(T_i, T_{ii}, T_{iii}) show a progressive reduction in pipeline completion time.

a rank fails, resizing increases the micro batch size on the affected pipeline stage; this extends that stage's step time , turning it into the pipeline's bottleneck. Proportionally larger micro batches also inflate activation footprints, introducing OOM risk.

DP&PP Domain Schedule. The root cause is that a rank failure perturbs the global balance of work and memory across the entire pipeline. Purely local adjustments within the failed stage's data-parallel (DP) group are therefore insufficient. Therefore, ElasWave's Graph Planner augments DP-domain scheduling with pipeline-parallel (PP)–domain rebalancing: in addition to resizing micro batches, it reshards PP stages to restore end-to-end balance. As shown in Figure 3②, ElasWave allows *migrating model layers* across stages to reshard stage workloads in the PP domain. By shifting a subset of layers out of the failure-impacted stage, we (i) reduce its excess compute and hence mitigate straggling, and (ii) reduce its extended activation footprint to lower the memory pressure.

Cost model. To determine the optimal assignment of layers to stages, we first formulate a cost model for the mini-step time on each stage. This allows us to cast the problem as a constrained minimax partition optimization. A *mini-step* denotes the forward backward process of one micro batch in the pipeline. For a given stage i, which hosts a set of ℓ_i layers and processes a local micro batch of size m_i, the mini-step time is modeled as:

$$\begin{aligned} &T_i^{\text{mini-step}}(\ell_i, m_i, r_{i-1}, r_i, r_{i+1}) \\ &= T_i^{C,\text{f}}(\ell_i, m_i) + T_i^{C,\text{b}}(\ell_i, m_i) \\ &\quad + \left[T_i^{\text{P2P,f}}(m_i, r_i, r_{i+1}) - \sigma_i^{\text{f}} T_i^{C,\text{f}}(\ell_i, m_i)\right] \\ &\quad + \left[T_i^{\text{P2P,b}}(m_i, r_{i-1}, r_i) - \sigma_i^{\text{b}} T_i^{C,\text{b}}(\ell_i, m_i)\right] \end{aligned} \tag{1}$$

Here, $T_i^{C,\text{f}}$ and $T_i^{C,\text{b}}$ are forward/backward compute times for stage i given ℓ_i and m_i; $T_i^{\text{P2P,f}}$ and $T_i^{\text{P2P,b}}$ are the point-to-point activation/gradient transfer times from $i \to i{+}1$ and $i{-}1 \to i$, respectively. The terms $\sigma_i^{\text{f}}, \sigma_i^{\text{b}} \in [0, 1]$ capture the overlap between compute and communication (a fraction of compute that hides communication). The quantities r_{i-1}, r_i, r_{i+1} parameterize the active ranks on adjacent stages that affect achievable P2P throughput (e.g., fan-in/out or contention). In practice, $T_i^{C,\cdot}(\ell_i, m_i)$ are profiled offline for relevant (ℓ_i, m_i) pairs before training; P2P times are predicted from m_i and hardware bandwidth; overlap coefficients $\sigma_i^{\cdot}$ are empirically profiled once and reused.

Solver. Given this cost model, the planner aims to find a layer assignment that satisfies: (i) is feasible under per-stage memory capcity cap_i (including parameter, optimizer state, and activation memory), and (ii) minimizes the worst per-stage mini-step time, i.e.,

$$\begin{aligned} &\min_{\{\ell_i\}} \max_i T_i^{\text{mini-step}}(\ell_i, m_i, \cdot) \\ &\text{s.t. } \text{Mem}(\ell_i, m_i) \le \text{cap}_i, \ \forall i. \end{aligned} \tag{2}$$

Algorithm 1 Minimax Layer Partition (DP)

Require: L = #layers; P = #stages; per-stage memory caps cap$[1..P]$; memory segment cost Mem$[u..v]$; mini-step cost $t_p([a..b]) = T_p^{mini-step}(b-a)$

Ensure: $f[P, L]$ = optimal worst-stage mini-step time over all feasible P-way partitions; $\{b_1, \ldots, b_{P-1}\}$ = right boundaries (stage j runs $(b_{j-1}{+}1)..b_j$, with $b_0{=}0, b_P{=}L$)

Notation: k = split index; $k^*(p, \ell)$ = optimal split for state (p, ℓ)

for $\ell = 1..L$ **do**
  $f[1, \ell] \leftarrow t_1([1..\ell])$
end for
▷ Transition
for $p = 2..P,\ \ell = p..L$ **do**
  $k^*(p, \ell) \leftarrow \arg\min_{k \in [p-1, \ell-1]} \max\{ f[p-1, k],\ t_p([k{+}1..\ell]) \}$
    s.t. $\text{Mem}[k{+}1..\ell] \le \text{cap}[p]$
  $f[p, \ell] \leftarrow \max\{ f[p-1, k^*(p, \ell)],\ t_p([k^*(p, \ell){+}1..\ell]) \}$
end for
$b_{P-1} \leftarrow k^*(P, L)$; for $p = P{-}1$ down to 2: $b_{p-1} \leftarrow k^*(p, b_p)$
return $f[P, L]$ and $\{b_1, \ldots, b_{P-1}\}$

We solve this partitioning problem using dynamic programming over contiguous blocks of layers. Let the layers be indexed from 1 to L and stages from 1 to P. For a contiguous layer block $[a..b]$ assigned to stage p, where $\ell_p = b-a+1$, we define its mini-step cost as $t_p([a..b]) \doteq T_p^{\text{mini-step}}(\ell_p, m_p, \cdot)$. This assignment is *feasible* only if its memory footprint $\text{Mem}(\ell_p, m_p)$ does not exceed the stage's capacity cap_p.

The DP state, $f[p, \ell]$, stores the optimal minimax mini-step time for partitioning the first ℓ layers ($[1..\ell]$) across the first p stages. The recurrence relation is formulated by choosing a split point $k \in [p-1..\ell-1]$ that partitions the layers into $[1..k]$ and $[k+1..\ell]$, minimizing the maximum cost for the two subproblems:

$$f[p, \ell] = \min_{k \in [p-1..\ell-1]} \max\{f[p-1, k], t_p([k+1..\ell])\} \quad (3)$$

After computing the DP table, we backtrack to find the optimal stage boundaries $\{b_1, \ldots, b_{P-1}\}$. The complete algorithm is detailed in Alg. 1.

The DP approach first guarantees memory feasibility (no OOM errors at any stage). and then optimizes the minimax objective, which directly targets pipeline throughput. To enable rapid decision-making at failure time, all required segment costs (Mem$[u..v]$ and $t_p([u..v])$) are precomputed from offline profiles and bandwidth models. While the theoretical complexity is $O(P\,L^2)$, the solver is efficient in practice due to aggressive pruning of infeasible partitions and the small number of stages (P) typical in pipeline parallelism.

By coupling micro batch resizing with layer re-partitioning under a principled minimax objective, our planner restores balance across both computation and memory and thereby recovers pipeline throughput while maintaining low MTTR.

4.3 DVFS Planner

Layer migration in the PP domain (Graph Planning) restores balance at the granularity of layers. However, the granularity of a layer can be coarser than the residual performance gap among stages after rebalancing. In such cases, any further layer migration would overshoot: the recipient stage would become the new straggler because a whole layer's worth of work exceeds the remaining bubble. This situation is illustrated in Fig. 4(b): the failure-impacted stage still exhibits mild straggling, yet moving another layer would invert the bottleneck. To eliminate these sub-layer-scale bubbles, we complement DP- and PP-domain scheduling with compute-unit scheduling via dynamic voltage and frequency scaling (DVFS).

Our policy is to up-clock only the residual straggler stage to shorten its mini-step time until it aligns with its peers, thereby removing the remaining bubble without perturbing the pipeline assignment. Because sustained high frequencies may accelerate hardware aging, we aim for the minimum necessary frequency increase.

Solver. Because higher frequency can stress hardware, we aim for the minimum necessary uplift. The controller proceeds in two steps (Alg. 2). First, it tests feasibility by setting the straggler to $f_{\max}$ and measuring its mini-step over a short observation window W. If even at $f_{\max}$ the stage still lags the target $T^\star$ (within tolerance ε), the gap is not compute-bound and is marked UNACHIEVABLE. Otherwise, alignment is possible; the controller runs a simple bisection between the current frequency and $f_{\max}$ to find the lowest frequency that meets $T^\star$ (respecting a minimum step $\Delta f_{\min}$).

Bubble-free restoration. By combining DP-domain resizing, PP-domain layer migration, and DVFS up-clocking, ElasWave removes the bubbles introduced by node failures across the pipeline. As shown in Fig. 4(c), the pipeline returns to a bubble-free status to maximize the throughput.

Algorithm 2 Minimum Bisection Frequency Scaling

Require: current frequency f_{cur}, maximum $f_{\max}$, target $T^\star$, tolerance ε, minimum step $\Delta f_{\min}$, Observation window W

Ensure: $(f^\star, \text{STATUS})$ with STATUS $\in$ {ACHIEVABLE, UNACHIEVABLE}

$t_{\text{cur}} \leftarrow \text{OBS\_TIME}(W)$
if $|t_{\text{cur}} - T^\star| \le \varepsilon$ **then return** $(f_{\text{cur}}, \text{ACHIEVABLE})$
end if
APPLY_FREQ$(f_{\max})$; $t_{\max} \leftarrow \text{OBS\_TIME}(W)$
if $t_{\max} > T^\star + \varepsilon$ **then return** $(f_{\max}, \text{UNACHIEVABLE})$
end if
Define evaluator $\mathcal{E}(f)$: APPLY_FREQ(f); OBS_TIME$(W) \le T^\star + \varepsilon$
$f^\star \leftarrow \text{BISECT\_MIN\_FEASIBLE}(f_{\text{lo}}{=}f_{\text{cur}},\ f_{\text{hi}}{=}f_{\max},\ \mathcal{E},\ \Delta f_{\min})$
return $(f^\star, \text{ACHIEVABLE})$

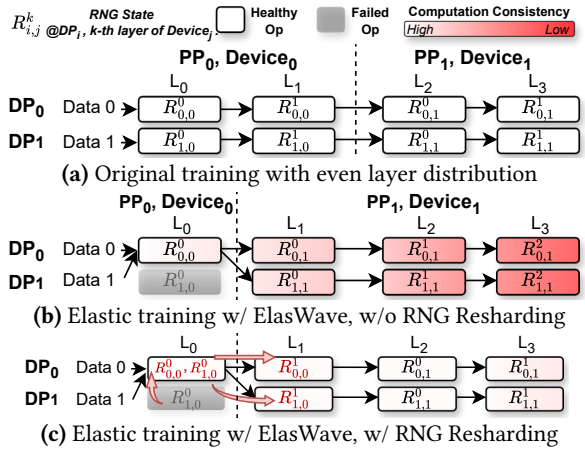


Figure 5. Elastic training without (b) or with (c) RNG Resharding. In (b), L_1 in PP_0 will be transferred to PP_1, the RNG state $R^0_{0,1}$ and $R^0_{1,1}$ in PP_1 will be directly applied to L_1, which introduces inconsistency. Besides, Data 1 is allocated to DP_0, but will be processed by RNG state $R^0_{0,0}$, which is also inconsistent with (a). In (c), RNG state $R^0_{1,0}$ will also be saved in DP_0, and will be applied to Data 1. After processing L_0, $R^1_{0,0}$ and $R^1_{0,1}$ will be sent to PP_1 of DP_0 and DP_1, respectively, and be used to process the trasferred L_1. Therefore, (c) achieves a similar convergent behavior.

4.4 RNG Resharding

To improve numerical consistency of elastic training, we propose the RNG (Random Number Generator) Resharding for random operations, such as dropout. In the distributed training, each node has an RNG to generate random number for random operations. As Figure 5 (b) shows, both layer rebalance and micro batch rebalance can change the RNG state for a specific data, which will further change the consistent behavior. Therefore, we propose the RNG Resharding in Figure 5 (c), which contains two steps for the RNG consistency.

In the layer rebalance, several layers in the failed stage will be transferred to other stages to avoid the affect of stragglers. Correspondingly, we need to transfer the RNG state from the failed stage in every forward propagation, and only apply the transferred RNG state to the transferred layers.

In the micro batch rebalance, data in the failed node will be dispatched to other nodes in the same stage. The dispatched data are supposed to be processed with their original RNG state. Therefore, every node needs to backup all RNG state of other nodes in the same stage, and uses corresponding RNG state to process the dispatched data.

In this way, we can ensure all data are processed with the same RNG state as that in the normal training, and achieve a same convergent behavior. Besides, we adjust the computation of average gradient in the global batch, so that the unevenly divided micro batch will not affect the final gradient results. We also notice that the change of the float-point addition order may introduce a small difference in the elastic schedule. However, in our evaluation in Section 7.5 , we find that it will not affect convergence consistency or cause loss spikes.

5 Parameter Fabric: Snapshot & Live Remap

5.1 Per-step Snapshot

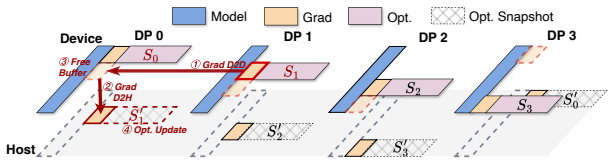
While ZeRO's memory efficiency makes it a default for large model training, its partitioned-state design fundamentally conflicts with elastic scaling by removing data redundancy. Our per-step snapshot mechanism provides fault tolerance for ZeRO-based training with minimal overhead.

In a standard ZeRO setup, each worker's GPU holds a unique partition of the optimizer states, which we denote as O_i^{device}. To introduce redundancy, we implement a ring-based snapshot scheme, as shown in Figures 6a. Each worker i becomes responsible for backing up the optimizer state partition from its neighbor, worker $(i+1) \bmod n$. This snapshot, denoted as O_i^{host}, is stored in worker i's host memory.

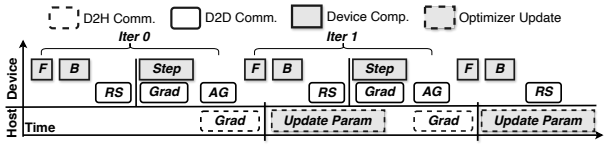
The design for achieving minimal overhead is illustrated in Figure 6b. The key principle is to make the snapshot process asynchronous and communication-efficient. Instead of transferring bulky optimizer states, we only transmit compact gradient shards to a peer worker, reducing snapshot communication by at least 4x for a mixed-precision optimizer like Adam. The actual parameter update for the snapshot is offloaded to the host CPU. By decoupling the resources (device vs. host) and execution timelines, the entire snapshot operation is performed in the background and overlaps with the next training iteration's computation. This ensures fault tolerance with negligible performance impact, as the critical path of training is not stalled.

5.2 Live Remap

Live Remap orchestrates the redistribution of optimizer states in response to any scaling event. Upon any scaling event, Live Remap initiates a four-step process. First, for scale-downs, an ①*Integrity Check* (Figure 7a) identifies failed workers (e.g., $F = \{2\}$) and confirms their state is recoverable from remaining on-device (O_i^{device}) and snapshot (O_i^{host}) partitions. Next, the system ② *Computes a Transfer Plan*. It creates the consolidated partitions ($O_{i,\text{consolidated}}$) as the logical union of all available on-device and host-snapshot data. It then computes an overlap matrix (M_{overlap}) by intersecting these source partitions with the final target partitions ($O_{j,\text{target}}$), defining the precise data flow (Figure 7b). The plan is executed in ③ *Opt. Redistribution* via D2D and H2D communication. Finally, in ④ *Finalization*, workers reconstruct new states and free unused memory.

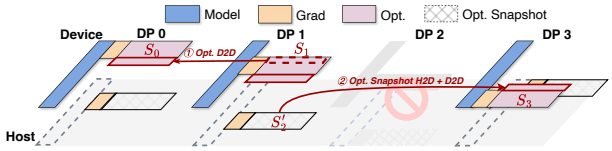


(a) Data Flow: Each worker (e.g., DP 1) sends the gradient shard for its local partition (S_1) to its peer (e.g., DP 0) via D2D (①). DP 0 then offloads this gradient to host memory (② D2H), frees the device buffer (③), and its host CPU updates the corresponding snapshot optimizer state on the host (S'_1) (④).

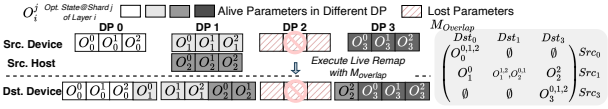


(b) Timeline: After Forward (F), Backward (B), and Reduce-Scatter (RS) passes, the D2D gradient transfer (Grad) for snapshotting occurs, running parallel to the optimizer update (Step). The gradient is then offloaded to the host (D2H Grad), overlapping with All-Gather (AG). The host's parameter update (Update Param) is hidden by the next iteration's computation, keeping the critical path clear.

Figure 6. The asynchronous per-step snapshot mechanism.



(a) Data Flow: When worker DP 2 fails, its state is recovered from snapshots on other workers (e.g., DP 1). Guided by the overlap matrix (M_{overlap}), worker DP 0 pulls necessary shards from DP 1's device via D2D (①), while DP 3 retrieves its required shards from DP 1's host snapshot (S'_1) through H2D+D2D (②).



(b) Remap Plan (M_{overlap}): The matrix defines the data transfer plan. For instance, the entry at (Src_1, Dst_0) indicates that partition O_1^0 must be transferred from worker 1 to worker 0, while diagonal entries like (Src_0, Dst_0) represent data that remains local.

Figure 7. Resharding process for a scale-down event.

6 MTTR Minimization

6.1 Dynamic Communicator

Modern training jobs face frequent hardware failure, but traditional distributed frameworks assume fixed resources and static communication domains, thus often force costly communicator reinitialization, incurring high overhead and degraded performance under resource change. NCCL exposes a communicator shrink API, but it essentially rebuilds a new communicator from the surviving ranks, still paying heavy reinitialization cost [4].

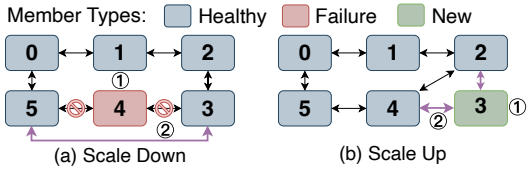


Figure 8. Dynamic Communicator Operations: (1) Scale-down: When an error is detected on a rank/node, it is removed from the worker pool to allow for training to continue, and only affected communicators are adjusted, while reusing existing links. (2) Scale-up: When a new worker joins the pool, only specific communicators are adjusted, and existing links are reused.

To address this limitation, we introduce the *Dynamic Communicator*, an adaptive framework that *scales up or scales down* (Figure 8) in real time to respond to changes in resource allocation by smoothly adapting communication interfaces. We adapt communicators online by *reusing existing links* and modifying only the affected groups. When resources change, the system creates only missing connections and preserves intact ones, avoiding global rebuilds and enabling recovery without a full restart.

Scalability. This approach provides (i) adaptive communication management for scale-up/down, (ii) efficient failure handling w/o full restarts, and (iii) efficient link management by creating only missing connections during dynamic operations. ❶ *Scale-down:* upon detecting a failed rank, the worker is removed and neighbors are reconnected to survival links (①), while only the necessary local communicators are updated (②). ❷ *Scale-up:* when a new worker joins, it establishes just the additional links (①–②); existing links and unrelated communicators are reused.

This *in-place, incremental* optimization eliminates cluster-wide rebuilds, *flattening the recovery cost with respect to communication scale* and rendering *MTTR a constant, sub-second bound.*

6.2 Model Recovery Acceleration

A straightforward method is **Blocked Layer Migration**, which copies the migrating layer to its new stage, after which resumes training. The stall scales with payload and bandwidth; when moves are frequent or span multiple layers, these stalls accumulate directly into MTTR.

Async. resharding with gradient precomputation. Overlapping the copy with training avoids the stall, but if the target processes micro-batches *before* the layer arrives, it cannot contribute that layer's gradients for those micro-batches, breaking gradient accumulation and forcing a later pause. Our method keeps the overlap and preserves accumulation (Fig. 9). While the target proceeds, the source runs a shadow

instance of the migrating layer for the early micro-batches (mb[0..k]), accumulates their missing gradients, and asynchronously ships this "payback" gradient to the target. The target merges it with local contributions once parameters are loaded. The only added cost is one gradient transmission per move, scheduled at lower priority and overlapped with ongoing compute, yielding non-blocking migration with complete gradient accumulation.

6.3 ZeRO Optimizer Recovery Acceleration

Contiguous Assignment and Intra-Stage Resharding. Migrating the optimizer state O_i of layer i from pipeline stage S to S+1 under the Contiguous assignment triggers all-to-all(v) resharding within both the source and destination DP groups, which dominates migration time. In this layout, each DP group maintains a single global byte array, and the ownership invariant requires each rank to hold one contiguous block of approximately equal size. After exporting O_i (Fig. 10a), the new cut points shift by $\approx |O_i|/D$ across the group, so multiple original intervals overlap each target interval; restoring contiguity therefore requires many-to-many personalized exchanges across ranks. In the figure, green arrows denote intra-stage exchanges and the dashed regions indicate released bytes once the layer has moved.

Interleaved Assignment and Point-to-Point Migration. Under the Interleaved assignment, migration reduces to D disjoint rank-to-rank sends of the layer's shards and eliminates any intra-stage resharding. Each layer is uniformly partitioned so that DP rank j always owns O_i^j for every layer. Consequently, moving O_i from stage S to S+1 consists only of sending O_i^j from rank j at stage S to rank j at stage S+1 (Fig. 10b); no stage-internal reshaping is needed.

Communication Cost. The Contiguous migration comprises a cross-stage transfer of $|O_i|$ plus intra-stage resharding that can be executed in $D-1$ neighbor rounds with cost

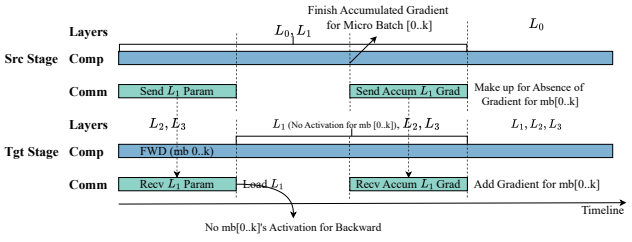
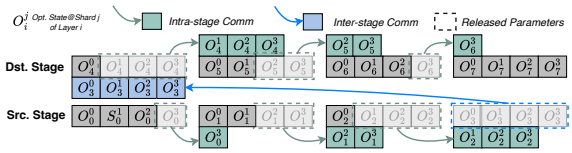
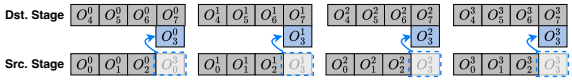


Figure 9. Asynchronous layer migration with gradient precomputation. While L_1 parameters stream to the target, the target proceeds with forward for micro-batches 0..k through its other layers, so no L_1 activations exist for their backward. The source keeps a shadow instance of L_1, completes backward for 0..k, accumulates the missing L_1 gradients, and asynchronously sends them to the target, which merges them to obtain a complete gradient without blocking.



(a) Contiguous: When a layer's optimizer state moves, its constituent shards (O_i^j, the j-th shard of layer i's state) are transferred from the source (down) to the destination (up). This triggers an all-to-all re-sharding within both stages, shown by green arrows, to restore a contiguous layout for the remaining shards.



(b) Interleaved: The optimizer state is partitioned such that rank j owns the j-th shard (O_i^j) for every layer. Migration thus reduces to direct point-to-point transfers, where each rank j sends its shard to the corresponding rank j in the destination stage, eliminating any intra-stage re-sharding.

Figure 10. Comparison of optimizer parameter migration under Contiguous vs. Interleaved ZeRO assignments.

$\frac{D-1}{2}|O_i|$. The total is therefore $\frac{D+1}{2}|O_i|$ bytes. In contrast, Interleaved performs exactly D 1:1 sends summing to $|O_i|$ bytes.

The change of ZeRO is purely an ownership-layout transformation: optimizer semantics and updates are unchanged, and each O_i is reconstructed from its shards exactly as in standard ZeRO.

7 Evaluation

7.1 Experimental Setup

We conduct experiments on a 12-node Ascend cluster, where each node has 8× Ascend 910B NPUs (32 GB memory) connected via 200 Gbps RoCE links; the software stack is CANN 8.0RC.3; the NPUs' initial frequency is 1,400 MHZ, with a maximum frequency of 1,650 MHZ. We compare ElasWave w ith two state-of-the-art systems: TorchFT, a widely-used industrial solution with DP-replica granularity elasticity, where an entire replica is removed on failure; and ReCycle, the academic state-of-the-art using data rerouting, which reroutes micro-batches within a DP domain to fit into pipeline bubbles and avoid straggling. However, vanilla 1F1B has become a weak baseline due to its low MFU and large bubble size. For fair comparison, we choose an SOTA pipeline on 1F1B: AdaPipe [55], which finds an optimal initial layer distribution to create a bubble-less schedule with maximized MFU. Our workloads consist of three Llama 2 models, with detailed configurations in Table 2.

7.2 Throughput Under Fail-stop Failures

Across models and shrink magnitudes, throughput orders ElasWave > ReCycle > TorchFT (Fig. 11). TorchFT is worst because each shrink drops whole DP replicas, yielding idle

Table 2. Workload Configurations for Llama 2 Models.

Model	Parallelism (TP,PP,DP)	Micro-batch Size	Global Batch Size
Llama2-7B	(4, 3, 8)	4	8192
Llama2-13B	(4, 6, 4)	2	2048
Llama2-34B	(4, 8, 3)	1	768

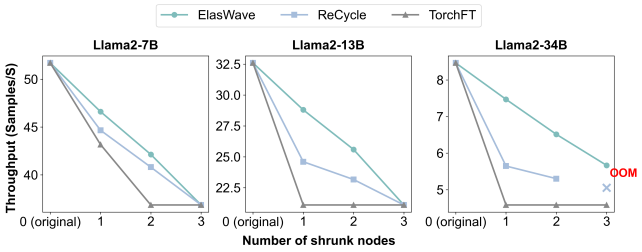
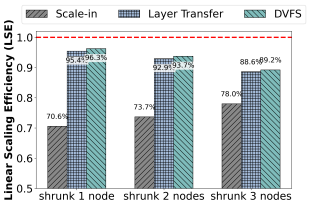
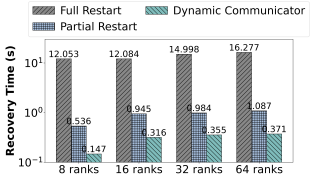


Figure 11. Throughput compairson under fail-stop failures.

capacity and cliff-like losses. ReCycle reroutes all failed work within a single PP stage; with many micro-batches the bubble budget is insufficient, creating stage stragglers—hence a sharp drop at the first shrink, smaller additional losses thereafter, and an OOM at Llama2-34B with three-node loss due to deferred weight-gradient memory. ElasWave spreads the failed load globally via graph migration, so throughput degrades nearly linearly. For example, on Llama2-34B with one node shrink it shows 60% higher throughput than TorchFT and 35% higher than ReCycle. When the lost NPUs equal an integer multiple of the DP-replica size, ReCycle and ElasWave degenerate to TorchFT (e.g., Llama2-7B at 3 nodes shrink, Llama2-13B at at 3 nodes shrink).



(a) Throughput efficiency (LSE) gains from each optimization under 1, 2, and 3 node failures.



(b) MTTR for three recovery scenarios: Full Restart (Grey), Partial Restart (Blue), and our Dynamic Communicator (Green).

Figure 12. ElasWave's performance under failures, showing (a) the breakdown of throughput efficiency gains and (b) the communication group recovery time (MTTR).

Throughput Breakdown. We use Linear Scaling Efficiency (LSE) to attribute gains across ElasWave's optimization steps. LSE serves as a "performance score," where a higher value indicates that throughput degradation upon node loss is closer to the ideal linear scaling, signifying less wasted computational power. Figure 12a presents a breakdown study isolating each optimization's contribution. The baseline scale-in policy absorbs a failed node's workload locally within its pipeline stage, creating a persistent straggler that gates throughput and yields low Load Scaling Efficiency (LSE). The most critical optimization, layer migration, resolves this bottleneck by globally redistributing the excess load across data and pipeline parallel dimensions, delivering a dominant improvement in LSE. This global rebalancing is the primary source of performance gain, though its benefits diminish as failures accumulate. Subsequently, DVFS provides fine-tuning by selectively up-clocking residual slow stages, adding another percentage point to the LSE. Combined, these optimizations achieve an LSE of ≥ 0.89 at all shrink points, demonstrating near-linear throughput. Layer migration accounts for 80–95% of the total improvement, with DVFS correcting minor residual imbalances.

7.3 Overhead of Per-step Snapshot

Table 3. Throughput with/without per-step snapshot (Samples per Second).

Model	No snapshot	With snapshot	Throughput loss (%)
Llama2-7B	51.941	51.700	0.46
Llama2-13B	32.805	32.602	0.62
Llama2-34B	8.545	8.487	0.69

Per-step snapshotting incurs a negligible performance overhead, with a throughput reduction of less than 1% across Llama-2 models ranging from 7B to 34B parameters (Table 3). This efficiency stems from a pipelined design that hides snapshot latency within the training's critical path, as shown in Figure 6b. Specifically, gradient transfers (D2D and D2H) are overlapped with concurrent device operations like the local optimizer step and All-Gather, while the final host-side parameter update is concealed by the subsequent training iteration. This ensures the backup process does not stall computation, and the overhead does not amplify with model size, demonstrating stable scaling.

7.4 MTTR Analysis

With online elasticity, most MTTR is eliminated and restart-related downtime disappears; the residual cost primarily comes from communicator reconstruction and system scheduling.

Communication Initialization. Current failure recovery relies on rebuilding communication groups—either a *full restart, which rebuilds the global communicator*, or a *partial restart, which rebuilds only the groups involving the failed node.* Both methods incur significant overhead. As shown in Figure 12b, our *Dynamic Communicator*'s localized design, which avoids costly global group reconstruction by only editing communication links adjacent to the failed rank, achieves a sub-second, near-constant recovery time of 0.15–0.37 s across 8–64 ranks. This yields speedup of 38–82×

over a full restart (12–16s) and 2.8–3.6× over a partial restart (0.54–1.09s).

Parameter Migration. Our optimizations for layer migration significantly reduce MTTR, with gains that amplify with model size (Figure 13). The total MTTR is reduced by 6–14% for the Llama-2-7B model, 13–22% for 13B, and a substantial 43–51% for 34B.

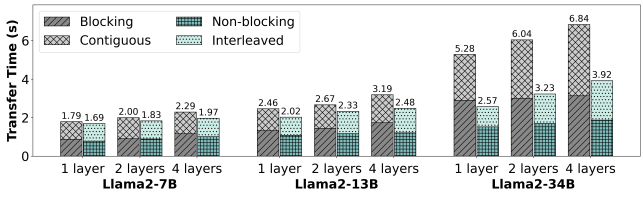


Figure 13. MTTR for layer migration on Llama-2 models, comparing our optimized design (non-blocking parameter migration, interleaved ZeRO layout) against a baseline (blocking copy, contiguous layout) when moving 1, 2, or 4 layers.

Model Parameter Migration. Our non-blocking approach reduces model parameter migration time with benefits that amplify at scale (Figure 13). The average MTTR reduction grows from a modest 8% for Llama-2-7B to 22% for 13B and 44% for 34B. This scaling advantage occurs because for larger models, data migration time dominates fixed orchestration costs. Our non-blocking design mitigates this dominant cost by overlapping the data transfer with other critical-path operations, effectively hiding the latency.

Optimizer Parameter Migration. For optimizer states, our interleaved ZeRO layout is 1.8–2.3× faster than the contiguous baseline on the 34B model (Figure 13). This advantage comes from converting the migration into parallel, rank-to-rank sends, which avoids the costly re-sharding and data compaction of the contiguous layout. The result is reduced communication volume and fewer network hotspots, making it essential for efficient elasticity at scale.

7.5 Convergence Consistency

In experiments, we evaluate the effectiveness of RNG Resharding on the convergence consistency. To do so, we fine-tuned a Llama2-7B model with LoRA on the GSM8K dataset, using an 8-NPU setup (TP=1, PP=4, DP=2) that scaled down to 7 NPUs to simulate a failure. We first measured the average loss difference ($\mathbb{E}_{\text{step}}[|Loss_{\text{normal}} - Loss_{\text{elastic}}|]$) between this elastic run and a no-failure baseline. Without RNG Resharding, the average loss difference was 0.2%, which dropped to a mere **0.045%** with our method, reducing 78% of the deviation. This improved training stability translates directly to better downstream task performance (Table 4). Averaged across all benchmarks, the absolute difference of accuracy deviation from the no-failure baseline was **45%** lower with RNG Resharding, confirming its effectiveness in preserving model quality during elastic scaling.

Table 4. Downstream task results with and without RNG Resharding. The Reduction is calculated as $1 - |\text{diff}|_{\text{RNG}}/|\text{diff}|_{\text{no-RNG}}$.

Task	Original Perf.	Perf. w/o RNG Resharding		Perf. w/ RNG Resharding		
	acc	acc	|diff|	acc	|diff|	**Reduction**
MMLU [15]	46.18	46.03	0.15	46.20	**0.02**	**86.7%**
BoolQ [6]	78.13	78.38	0.25	78.32	**0.19**	**24.0%**
BBH [54]	35.90	36.20	0.30	35.70	**0.20**	**33.3%**
AGIEval [70]	23.91	24.08	0.17	24.00	**0.09**	**47.1%**
CEval [17]	34.03	34.92	0.89	34.62	**0.59**	**33.7%**
Average	-	-	-	-	-	**45.0%**

7.6 End-to-End Performance

On real spot instance traces [38], ElasWave consistently achieves the highest time-averaged throughput, outperforming ReCycle by 10–20% and TorchFT by 50–70% across all models and traces. This advantage stems from its coordinated DP+PP rebalancing, which restores near-linear steady states after capacity changes. TorchFT's full restarts lead to a long MTTR ($\tilde{2}$0s) and the lowest performance. This ranking proves robust across diverse trace patterns.

7.7 Case Study

We present two case studies showing ElasWave's elasticity under both fail-slow conditions and expert-parallel MoE workloads, consistently restoring throughput via rapid rebalancing after perturbations.

Fail-slow Mitigation. We demonstrate ElasWave's effectiveness in mitigating transient hardware slowdowns (stragglers). We simulate three straggler levels—Low, Medium, and High—by artificially slowing down one worker. As shown in Figure 15a, the straggler degrades the normalized throughput to 0.931, 0.865, and 0.809, respectively. By dynamically rebalancing the workload, ElasWave recovers the throughput to 0.951, 0.937, and 0.905. This corresponds to recouping over 50% of the performance loss in the medium and high straggler scenarios, showcasing its capability to maintain high efficiency under heterogeneous hardware conditions.

Expert Parallelism. We also evaluate ElasWave on a Mixture-of-Experts (MoE) model using Llama2-13B, where elasticity is critical for managing expert capacity. In a failure scenario, we compare ElasWave against a baseline framework, TorchFT. After a failure, the baseline's throughput drops to 9.92 samples/sec from an initial 15.73 samples/sec. In contrast, ElasWave's efficient recovery and rebalancing mechanisms achieve a throughput of 13.13 samples/sec, a 32% improvement over TorchFT, recovering a significant portion of the lost performance. This demonstrates ElasWave's superior capability in complex, dynamic workloads like MoE.

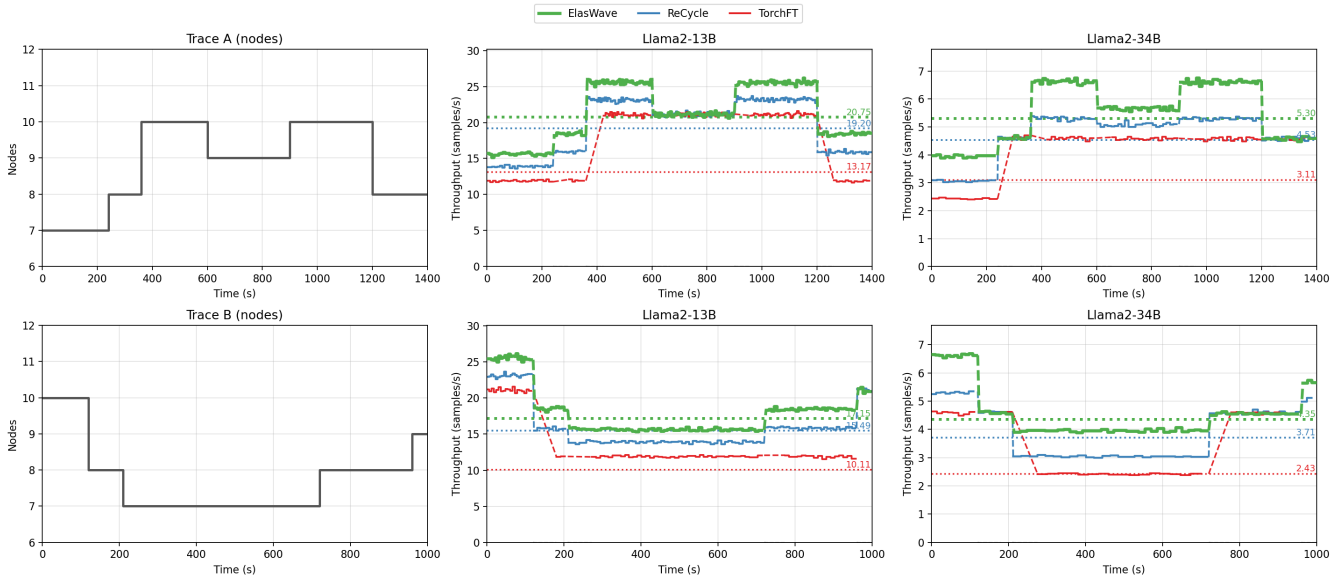
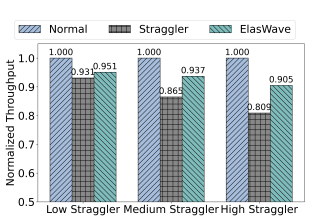
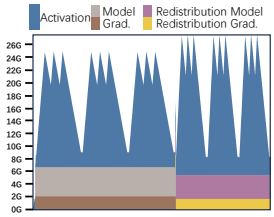


Figure 14. Throughput of ElasWave (Green), ReCycle (Blue), and TorchFT (Red) on two real-world spot instance traces. Trace A is plateau-heavy and Trace B is shrink-heavy.



(a) Straggler mitigation under three simulated slowdown levels (Low, Medium, High).



(b) Per-rank memory footprint increases upon cluster shrink due to workload redistribution.

Figure 15. ElasWave performance analysis in case study.

7.8 Discussions

ElasWave delivers robust application-level elasticity, handling failures and stragglers by optimizing workload distribution to ensure training continuity. While it masterfully mitigates the *impact* of systemic issues like network congestion, addressing their root cause is beyond its scope. This deliberate focus allows ElasWave to excel within its application-level domain, providing a resilient core for future co-design with network-aware infrastructure.

Extensibility. The framework's flexibility is further validated on MoE models. That our general-purpose elastic engine functions effectively in such a specialized, dynamic domain is a testament to its robust design. While adapting to SOTA MoE systems like DuoPipe remains future work, our solution provides a strong and versatile foundation for it.

OOM Risks. ElasWave successfully navigates the fundamental memory trade-offs of elasticity (Figure 15b). An increased per-rank memory footprint upon cluster shrink is an unavoidable consequence of workload redistribution. ElasWave adeptly manages these dynamics, absorbing transient activation spikes and guaranteeing against OOM failures. The resulting stable, albeit elevated, memory profile reflects this robust resource management and opens avenues for future work in advanced offloading schemes.

8 Related Work

Checkpoint-based fault tolerance. *Checkpointing* enables recovery from the most recent stable state by periodically saving the training state. Recent work focuses on mitigating the significant overhead of this process, including asynchronous [2, 36, 42, 59], lightweight in-memory [60], incremental checkpoints [1, 8], and adaptive adjustment of checkpoint frequency [25, 35, 39, 66]. To minimize MTTR, ElasWave skips checkpoint-based rollback and performs online elastic recovery from in-memory step state.

Elastic training. ElasWave targets *efficient elastic execution under failures* that natively satisfies multiple metrics and jointly minimizing MTTR and maintaining high post-change throughput while preserving statistical validity.

MTTR-oriented recovery. Mainstream toolchains implement elasticity as *restart-and-resume* from checkpoints or in-process state commits (coarse job-level granularity); in-memory checkpointing reduces I/O but still triggers broad pauses and communicator rebuilds, yielding only coarse control over MTTR and post-recovery throughput [46, 50, 53, 60]. Online reconfiguration for anticipated/detected failures (e.g., spot/preemptible) improves system availability, but MTTR

remains coupled to cluster scale and the chosen granularity [9, 21, 51, 57].

Throughput-oriented elasticity. Prior work boosts *steady-state throughput* by reshaping parallelism or injecting redundancy under resource changes or failures. Approaches range from job/stage template switching (pipeline templates) to DP-group rerouting and micro-batch rebalancing; typical granularities are job-level [3], stage/pipeline-level [21, 57], DP-group-level [51], and micro-batch-level [9]. These methods sustain progress but can incur overhead in failure-free periods (redundant compute) and suffer MTTR degradation under heterogeneity or frequent reconfiguration; they also often optimize only a *single* dimension (e.g., DP routing) rather than coordinating data/model/hardware jointly.

Convergence under elasticity. Some efforts preserve accuracy consistency during elasticity (e.g., DP resizing rules), while largely assuming *DP-only* elasticity and leaving cross-dimension rebalancing (DP/PP/micro-batch) unresolved [11, 20, 27, 28, 48].

Failure Detection. Effective *fault detection* is essential to minimize downtime and error propagation, leveraging tools like NVIDIA DCGM [43] and mechanisms such as timeouts, heartbeats, and log analysis [5, 12, 22, 26, 34, 68]. In addition, the ElasWave Agent includes hardware- and process-level detectors on Ascend clusters. Due to page limits, we omit details.

Resilience–Convergence Tradeoffs. Several fault-tolerant strategies trade statistical convergence for progress under failures or delays. *Local/periodic-averaging SGD* and federated variants reduce synchronization to mask stragglers, but client drift and heterogeneity yield slower or biased convergence and accuracy gaps [14, 23, 30].*Buffered asynchronous aggregation* admits clients as they arrive to sustain throughput, but the resulting asynchrony and client drift degrade statistical efficiency [41]. These trade-offs are misaligned with frontier-scale pretraining, where accuracy, stability, and reproducibility are first-class requirements. In contrast, ElasWave couples resilience scheduling with RNG-state consistency to maintain optimization fidelity despite failures.

9 Conclusion

We presented ElasWave, an elastic-native LLM training system that performs per-step fault tolerance via multi-dimen sional scheduling across dataflow, graph, DVFS, and RNG. The design couples an in-place dynamic communicator with non-blocking layer migration, interleaved ZeRO state movement, and snapshot-based live remap to meet the four production goals: parameter consistency, low MTTR, high post-change throughput, and computation consistency. The system is currently in a preliminary experimental stage and will undergo further evaluation at extensive scale. On our testbed, ElasWave improves throughput by up to 1.60x than TorchFT and up to 1.35x than ReCycle. Communicator recovery MTTR is improved by up to 82× and 3.6× compared with full and partial rebuilds. Non-blocking migration with interleaved ZeRO cuts Layer migration MTTR by up to 51% compared with blocking migration using default ZeRO. RNG Resharding reduces convergence deviation by 78%. These results demonstrate that ElasWave delivers fast, consistent, and scalable elasticity for large-model training and provides a practical path to robust pretraining on fluctuating, hyperscale clusters.

References

[1] Amey Agrawal, Sameer Reddy, Satwik Bhattamishra, Venkata Prabhakara Sarath Nookala, Vidushi Vashishth, Kexin Rong, and Alexey Tumanov. 2024. Inshrinkerator: Compressing Deep Learning Training Checkpoints via Dynamic Quantization. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 1012–1031.

[2] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: Scalable, Low-cost Training of Massive Deep Learning Models. In *EuroSys*. https://www.microsoft.com/en-us/research/wp-content/uploads/2022/03/varuna-eurosys22.pdf

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

[4] John Bachan, Kaiming Ouyang, Misbah Mubarak, Thomas Gillis, Bruce Chang, Devendar Bureddy, Giuseppe Congiu, Keith Caton, Kyle Aubrey, and Xiaofan Li. 2025. *Enabling Fast Inference and Resilient Training with NCCL 2.27*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/enabling-fast-inference-and-resilient-training-with-nccl-2-27/

[5] Marco Barletta, Marcello Cinque, Catello Di Martino, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2024. Mutiny! How does Kubernetes fail, and what can we do about it?. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–14.

[6] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 2924–2936. doi:10.18653/v1/N19-1300

[7] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[8] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 929–943.

[9] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs Using Pipeline Adaptation. In *SOSP*. https://dl.acm.org/doi/10.1145/3694715.3695960

[10] Google Cloud. 2025. *TPU v6e*. https://cloud.google.com/tpu/docs/v6e?hl=zh-cn Last updated (UTC).

[11] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[12] Wei Guan, Jian Cao, Shiyou Qian, Jianqi Gao, and Chun Ouyang. 2024. Logllm: Log-based anomaly detection using large language models. *arXiv preprint arXiv:2411.08561* (2024).

[13] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. 2024. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1110–1125.

[14] Farzin Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Viveck Cadambe. 2019. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. *Advances in Neural Information Processing Systems* 32 (2019).

[15] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. *Proceedings of the International Conference on Learning Representations (ICLR)* (2021).

[16] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 709–729.

[17] Yuzhen Huang, Yuzhuo Bai, Zhihao Zhu, Junlei Zhang, Jinghan Zhang, Tangjun Su, Junteng Liu, Chuancheng Lv, Yikai Zhang, Jiayi Lei, Yao Fu, Maosong Sun, and Junxian He. 2023. C-Eval: A Multi-Level Multi-Discipline Chinese Evaluation Suite for Foundation Models. *arXiv preprint arXiv:2305.08322* (2023).

[18] Huawei. 2025. *Huawei Unveils World's Most Powerful SuperPoDs and SuperClusters*. https://www.huawei.com/en/news/2025/9/hc-lingqu-ai-superpod News post.

[19] HUAWEI Developers. 2025. About the Service—CANN. https://developer.huawei.com/consumer/en/doc/hiai-guides/introduction-0000001051486804. Last updated: 2025-07-14; Accessed: 2025-09-26.

[20] Sung Hwang, Jaehee Kim, and Minsoo Ryu. 2021. Elastic Deep Learning with Dynamic Resource Allocation. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NeurIPS)*. 12345–12356. https://proceedings.neurips.cc/paper/2021/hash/elastic-dynamic-resource-allocation.html

[21] Insu Jang, Zhenning Yang, Zhen Zhang, et al. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *SOSP*. doi:10.1145/3600006.3613152

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

[23] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. 2020. Scaffold: Stochastic controlled averaging for federated learning. In *International conference on machine learning*. PMLR, 5132–5143.

[24] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. 2025. Revisiting Reliability in Large-Scale Machine Learning Research Clusters. arXiv:2410.21680 [cs.DC] https://arxiv.org/abs/2410.21680

[25] Hongliang Li, Zichen Wang, Hairui Zhao, Meng Zhang, Xiang Li, and Haixiao Xu. 2025. Convergence-aware optimal checkpointing for exploratory deep learning training jobs. *Future Generation Computer Systems* 164 (2025), 107597.

[26] Jie Li, Rui Wang, Ghazanfar Ali, Tommy Dang, Alan Sill, and Yong Chen. 2023. Workload failure prediction for data centers. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 479–485.

[27] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 835–850.

[28] Mingzhen Li, Wencong Xiao, Biao Sun, Hanyu Zhao, Hailong Yang, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. 2022. Easyscale: Accuracy-consistent elastic training for deep learning. *arXiv preprint arXiv:2208.14228* (2022).

[29] Mingzhen Li, Wencong Xiao, Biao Sun, Hanyu Zhao, Hailong Yang, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, Wei Lin, and Depei Qian. 2023. EasyScale: Accuracy-consistent Elastic Training for Deep Learning. arXiv:2208.14228 [cs.DC] https://arxiv.org/abs/2208.14228

[30] Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. 2019. On the convergence of fedavg on non-iid data. *arXiv preprint arXiv:1907.02189* (2019).

[31] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, et al. 2025. Understanding Stragglers in Large Model Training Using What-if Analysis. *arXiv preprint arXiv:2505.05713* (2025).

[32] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

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

[34] Heting Liu, Zhichao Li, Cheng Tan, Rongqiu Yang, Guohong Cao, Zherui Liu, and Chuanxiong Guo. 2023. Predicting GPU Failures With High Precision Under Deep Learning Workloads. In *Proceedings of the 16th ACM International Conference on Systems and Storage*. 124–135.

[35] Hangyu Liu, Shouxi Luo, Ke Li, Huanlai Xing, and Bo Peng. 2025. Checkflow: Low-Overhead Checkpointing for Deep Learning Training. *IEEE Computer Architecture Letters* (2025).

[36] Avinash Maurya, Robert Underwood, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Datastates-llm: Lazy asynchronous checkpointing for large language models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 227–239.

[37] Meta AI. 2025. *The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation.* https://ai.meta.com/blog/llama-4-multimodal-intelligence/ Blog post.

[38] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1112–1127.

[39] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. {CheckFreq}: Frequent,{Fine-Grained}{DNN} Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.

[40] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. arXiv:2104.04473 [cs.CL] https://arxiv.org/abs/2104.04473

[41] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. 2022. Federated learning with buffered asynchronous aggregation. In *International conference on artificial intelligence and statistics*. PMLR, 3581–3607.

[42] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 172–181.

[43] NVIDIA. 2025. NVIDIA DCGM. https://developer.nvidia.com/dcgm.

[44] OpenAI. 2025. *Introducing Stargate UAE.* https://openai.com/index/introducing-stargate-uae/ News post.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703

[46] PyTorch. 2025. Torch Distributed Elastic. https://docs.pytorch.org/docs/stable/distributed.elastic.html.

[47] Penghui Qi, Xinyi Wan, Nyamdavaa Amar, and Min Lin. 2024. Pipeline parallelism with controllable memory. *Advances in Neural Information Processing Systems* 37 (2024), 46539–46566.

[48] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *OSDI*. https://www.usenix.org/system/files/osdi21-qiao.pdf

[49] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[50] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 3505–3506.

[51] Tristan Rice and Howard Huang. 2025. Fault Tolerant Llama: training with 2000 synthetic failures every ~15 seconds and no checkpoints on Crusoe L40S. PyTorch Blog. https://pytorch.org/blog/fault-tolerant-llama-training-with-2000-synthetic-failures-every-15-seconds-and-no-checkpoints-on-crusoe-l40s/

[52] Harald Semmelrock, Tony Ross-Hellauer, Simone Kopeinik, Dieter Theiler, Armin Haberl, Stefan Thalmann, and Dominik Kowald. 2025. Reproducibility in machine-learning-based research: Overview, barriers, and drivers. *AI Magazine* 46, 2 (2025), e70002.

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

[54] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615* (2022).

[55] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. 2024. AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 86–100. doi:10.1145/3620666.3651359

[56] Qwen Team. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* 2 (2024).

[57] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 497–513.

[58] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. 2025. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *NSDI*. https://www.usenix.org/system/files/nsdi25-wan-borui.pdf

[59] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. 2024. Fastpersist: Accelerating model checkpointing in deep learning. *arXiv preprint arXiv:2406.13768* (2024).

[60] Junjia Wang, Yuqing Shen, Yu Lin, et al. 2023. Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP)*. arXiv:2302.00001.

[61] Qinlong Wang, Tingfeng Lan, Yinghao Tang, Ziling Huang, Yiheng Du, Haitao Zhang, Jian Sha, Hui Lu, Yuanchun Zhou, Ke Zhang, and Mingjie Tang. 2024. DLRover-RM: Resource Optimization for Deep Recommendation Models Training in the Cloud. arXiv:2304.01468 [cs.DC] https://arxiv.org/abs/2304.01468

[62] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.

[63] Tianyuan Wu, Lunxi Cao, Hanfeng Lu, Xiaoxiao Jiang, Yinghao Yu, Siran Yang, Guodong Yang, Jiamang Wang, Lin Qu, Liping Zhang, and Wei Wang. 2025. Adaptra: Straggler-Resilient Hybrid-Parallel Training with Pipeline Adaptation. arXiv:2504.19232 [cs.DC] https://arxiv.org/abs/2504.19232

[64] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2024. FALCON: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training. *arXiv preprint arXiv:2410.12588* (2024).

[65] xAI. 2025. *Grok 4*. https://x.ai/news/grok-4 News post.

[66] Chenxuan Yao, Yuchong Hu, Feifan Liu, Zhengyu Liu, and Dan Feng. 2025. LowDiff: Efficient Frequent Checkpointing via Low-Cost Differential for High-Performance Distributed Training Systems. *arXiv preprint arXiv:2509.04084* (2025).

[67] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. 2024. Hap: Spmd dnn training on heterogeneous gpu clusters with automated program synthesis. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 524–541.

[68] Zhiwei Zhang, Saifei Li, Lijie Zhang, Jianbin Ye, Chunduo Hu, and Lianshan Yan. 2025. LLM-LADE: Large language model-based log anomaly detection with explanation. *Knowledge-Based Systems* 326 (2025), 114064.

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

[70] Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2023. AGIEval: A Human-Centric Benchmark for Evaluating Foundation Models. arXiv:2304.06364 [cs.CL]