Phalanx: A Formally Verified, Topology-Aware Control Plane for Fault-Tolerant Training on Kubernetes GPU Clusters

Abstract

Large-scale training is now failure-dominated: Meta reported 419 unexpected interruptions across a 54-day Llama 3 run on 16,384 H100 GPUs, roughly one every three hours, and synchronous training semantics mean a single dead GPU can stall the entire job. The infrastructure that manages these jobs is split down the middle. Kubernetes-side schedulers (Volcano, Kueue, YuniKorn, NVIDIA KAI, and the alpha Workload Aware Scheduling primitives that only entered upstream Kubernetes in v1.35 and v1.36) decide where a gang of pods lands, with growing but incomplete topology awareness. Framework-side elasticity layers (TorchElastic, Oobleck, Bamboo, Parcae) decide how a job survives a failure, with no view of cluster topology and no ability to claim replacement hardware atomically. The seam between them is exactly where large runs bleed goodput: failure-triggered rescheduling is today neither topology-aware nor provably checkpoint-consistent. We present **Phalanx**, a Kubernetes-native control plane that unifies the two concerns. Phalanx contributes (1) HDF-B, a hierarchical deepest-fit gang placement algorithm with buddy-aligned allocation over a topology tree compiled from Dynamic Resource Allocation attributes, for which we prove per-gang bottleneck optimality, a factor-2 capacity augmentation bound, and a bounded failure blast radius, all three being per-placement guarantees rather than global multi-tenant claims, since we prove the global objective NP-hard; (2) the Rank Reformation Protocol (RRP), an epoch-fenced reconfiguration protocol whose safety (checkpoint consistency, exclusive placement, zombie fencing) we establish by inductive invariants over a TLA$^+$ specification and whose liveness we establish under partial synchrony, in the style of Anvil's eventually stable reconciliation; we mechanically check both with TLC across 834,050 distinct protocol states, and a mutation study shows the checker exhibiting a liveness counterexample the moment the protocol's contended-commit retry transition is deleted; and (3) an executed evaluation: beyond the TLC campaign, a full-scale discrete-event instantiation of the model at 131,072 leaves in which Replan answers node failures with a p99 latency of 0.9 microseconds while migrating exactly 8 ranks, recovery placements hold median step-communication inflation to 1.00× against 8.9× for topology-blind replacement on a hierarchical bandwidth model, and Monte Carlo failure injection over 1,000 54-day runs per configuration reproduces the renewal model within 1.4 points, yielding, under a recovery-latency model calibrated to published peer-RAM restore figures, effective training time ratios of 0.980 at 16K GPUs and 0.891 at 131K GPUs where the restart-based baseline measures 0.467, with a sensitivity sweep confirming that fast in-memory recovery dominates restart across the entire plausible range of the assumed parameters. The design is hardware-generation agnostic across Ampere, Hopper, and Blackwell because the topology tree is derived from device attributes rather than hard-coded fanouts.

1 Introduction: the reliability wall

Two curves are crossing. The first is cluster scale: frontier training runs moved from hundreds of GPUs to tens of thousands in about four years, and public roadmaps point at hundreds of thousands. The second is component reliability, which is not improving at anywhere near the rate scale is growing. The product of the two is a mean time between failures that shrinks linearly with GPU count. The best public datapoint is the Llama 3 herd report [1]: over a 54-day pre-training snapshot on 16,384 H100 GPUs, Meta logged 466 job interruptions, 419 of them unexpected, one failure roughly every 3 hours; 58.7% of the unexpected interruptions were GPU-related, with 30.1% attributed to GPU failures including NVLink and 17.2% to HBM3 memory, and the team still held effective training time above 90% with only three incidents needing significant manual intervention [1, 2]. Extrapolating the same per-device failure rate, a 131,072-GPU cluster faces an interruption every 23 minutes. Meta's broader reliability study of two research clusters, covering 11 months, four million jobs, and 150 million GPU hours, formalizes the productivity metric this paper optimizes, the effective training time ratio (ETTR), and shows why checkpoint cadence and restart latency dominate it at scale [3]. Earlier evidence says none of this is new: the Philly trace analysis of a multi-tenant Microsoft cluster already found DNN training jobs failing on average every 45 minutes [4].

Synchronous data-parallel training turns each component failure into a whole-job event. Every rank participates in every collective; one unresponsive NCCL peer stalls 16,383 others. The infrastructure response has been bifurcated in a way this paper argues is the core architectural mistake of the current stack.

The scheduling half. Kubernetes was built around one-pod-at-a-time scheduling for stateless services. Gang semantics, where a job needs all-or-nothing placement of N pods, arrived only through external schedulers: Volcano's PodGroups [27], Kueue's admission and quota layer [28], YuniKorn [29], and NVIDIA's KAI-Scheduler [30], which adds hierarchical PodGroups, topology-aware scheduling, and DRA support for GB200/GB300-class parts, with public scale tests at thousands of nodes. Upstream Kubernetes itself shipped alpha native gang scheduling only in v1.35 (KEP-4671), and the v1.36 release [31] extends it with a decoupled Workload and PodGroup API, workload-aware preemption, topology-aware workload scheduling (KEP-5732 [32]), DRA ResourceClaim support for workloads, and Job controller integration, all alpha and disabled by default behind feature gates including `GenericWorkload`, `GangScheduling`, and `WorkloadAwarePreemption`. Every one of these systems answers the question "where should this gang start" and none of them answers "what happens at failure number 217 of 419."

The elasticity half. TorchElastic [22] re-forms the process group through rendezvous and restarts from the last checkpoint. Oobleck [14] precomputes heterogeneous pipeline templates so a job can reshape onto fewer nodes without a full restart, with a combinatorial guarantee that some template combination covers the survivors, and it beats restart-based baselines such as Bamboo [18] and Varuna [19] by up to 13.9×. Gemini [15] checkpoints to peer CPU memory with a provably near-optimal placement strategy and recovers more than 13× faster than remote-storage checkpointing, at per-iteration cadence with no throughput overhead. Parcae [20] and ReCycle [21] extend the family toward preemption-heavy and spare-free environments. Every one of these systems answers "how do I keep computing after a failure" from inside the job, blind to the cluster: they cannot see NVLink domains or rail alignment, cannot atomically claim a spare node, and cannot prevent the cluster scheduler from handing their vacated GPUs to someone else mid-recovery.

The seam between the two halves is where correctness and goodput both leak. When a GPU dies today, the elastic layer detects it (slowly, via NCCL timeouts measured in minutes at default settings), tears down, re-rendezvouses, and asks the scheduler for capacity as if it were a brand-new job. The scheduler, which has no concept of "this is failure recovery for a job whose surviving 16,376 ranks occupy a specific rail-aligned footprint," places replacements wherever bin-packing lands them, frequently off-rail or across a spine, degrading every subsequent step. Meanwhile nothing in either layer proves that the state the job resumes from is a consistent checkpoint: consistency is a convention enforced by framework code paths that no one has verified. Formal verification has reached Kubernetes controllers, but not this problem: Anvil [7] machine-verified liveness, in the form of eventually stable reconciliation (ESR), for ZooKeeper, RabbitMQ, and FluentBit controllers using the Verus [8] SMT-based verifier for Rust, demonstrating the technique but never applying it to a scheduler or a training reconfiguration protocol.

Thesis. Failure recovery at 10,000-GPU scale *is* a scheduling event, and scheduling at that scale *is* a fault-tolerance protocol. A control plane that treats them as one problem makes failure-triggered rescheduling topology-aware (recovering placements as good as initial placements) and provably checkpoint-consistent (recovering states that are guaranteed, not assumed, to be valid), while retaining Kubernetes-native APIs and hardware-generation agnosticism.

Contributions.

- **C1.** A formal model of GPU cluster topology as a leveled tree compiled from Kubernetes Dynamic Resource Allocation (DRA) device attributes, making the same algorithms apply unmodified to 8-GPU HGX Ampere/Hopper nodes and 72-GPU Blackwell NVL72 domains (Sections 3, 4).
- **C2.** HDF-B, a gang placement algorithm combining hierarchical deepest-fit search with buddy-aligned carving, with three proved properties: per-gang bottleneck optimality (Theorem 2), a factor-2 capacity augmentation admission guarantee (Proposition 1), and a bounded failure blast radius (Proposition 2). We situate the multi-gang problem against known strong NP-hardness and approximation bounds from parallel-task scheduling and strip packing (Theorem 1).
- **C3.** The Rank Reformation Protocol (RRP), an epoch-fenced reconfiguration protocol layered on a linearizable store. We give a TLA$^+$ specification, prove three safety invariants by induction (checkpoint consistency, exclusive placement, zombie fencing), prove liveness under partial synchrony mapped onto Anvil's ESR shape, and mechanically check safety and liveness with TLC, including a mutation study in which the checker finds the exact contention livelock that the protocol's retry transition exists to prevent (Section 6).
- **C4.** Lifecycle unification: planned maintenance, defragmentation, and elastic scale changes are the same epoch transition as failure recovery, so one verified protocol covers the job's entire life, including disaggregated inference gangs (Section 7).
- **C5.** An executed evaluation combining the TLC verification campaign with a 131,072-leaf discrete-event testbed: sub-microsecond median reformation planning, 1.00× median recovery inflation versus 8.9× topology-blind, Monte Carlo ETTR within 1.4 points of the renewal model across six configurations, and 100,000-transition trace validation with zero divergences (Section 8).

2 Why the current stack cannot close the gap

2.1 Kubernetes' scheduling model versus gang semantics

The kube-scheduler binds pods one at a time. For a 2,048-pod training gang this creates deadlock and starvation modes that are well documented: partial placements hold GPUs hostage while waiting for stragglers, and two half-placed gangs can mutually starve. The scheduling framework's Permit plugin phase (used by the coscheduling plugin, Volcano, and KAI) approximates atomicity by holding pods at the gate until a quorum accumulates, but holding is not reserving: nothing prevents interleavings in which capacity evaporates between the gate opening and binding. The v1.35/v1.36 Workload Aware Scheduling work is Kubernetes' own acknowledgment of the mismatch: it introduces a Workload API and a runtime PodGroup object with a `gang.minCount` quorum, schedules the PodGroup in a dedicated cycle, and adds group-level preemption [31]. These primitives are alpha, disabled by default, and cover admission of homogeneous groups. Nothing in the upstream design addresses what happens when a member of a *running* gang dies; the group primitive covers admission, not survival.

2.2 Topology awareness is bolted on, and recovery ignores it

Modern training performance is a function of where ranks land relative to three physical tiers. Inside an HGX H100 node, each GPU has 900 GB/s of aggregate NVLink 4 bandwidth to its seven peers. Leaving the node, each GPU typically owns one 400 Gb/s NDR NIC, roughly 50 GB/s, an 18× cliff. In rail-optimized fabrics, NIC k of every server connects to leaf switch k, so communication between equal-indexed GPUs of different servers stays one hop inside a rail, while misaligned pairs traverse the spine. Placement that respects these tiers (tensor-parallel groups inside NVLink domains, data-parallel peers rail-aligned) is the difference between the fabric helping and the fabric being the bottleneck; Volcano's network-topology mode, Kueue's Topology Aware Scheduling, and KAI's TAS all exist because of this, and KAI's hierarchical PodGroups extend it to multi-component workloads in the Grove/Dynamo disaggregated-serving style [30].

Now observe what happens at failure time in every one of these systems: nothing topology-aware. The replacement pod arrives through the generic queue. Kueue's documentation notes DRA resources are not accounted in TAS capacity. Worse, correlated NIC and switch failures in rail-optimized fabrics create *topological fragmentation*, where surviving capacity is plentiful but rail-misaligned, precisely the regime where a topology-blind replacement is most damaging. Recovery placement is a harder instance of the placement problem (placement under a constrained residual topology with a warm-start footprint), and the current stack solves it with the least capable instrument it has.

System	Topo gang	Verified reconfig	K8s native	Elastic
Volcano / Kueue / KAI	yes	no	yes	partial
K8s WAS v1.35/1.36	partial	no	yes	no
Oobleck	no	combinat.	no	yes
Gemini	no	ckpt only	no	no
Singularity	no	no	no	yes
TorchElastic	no	no	partial	yes
Anvil	no	yes (ESR)	yes	n/a
Phalanx	**yes**	**yes (S+L)**	**yes**	**yes**

Table 1: No prior system satisfies all four properties. "S+L" denotes machine-checked safety and liveness.

2.3 Elastic frameworks cannot fix this from inside the job

Oobleck's pipeline templates are the strongest prior on the reshaping side, and its guarantee is real but combinatorial: given at least $f+1$ template-instantiated pipeline replicas, any f simultaneous failures leave a coverable configuration [14]. It is not machine-checked, it is not a cluster scheduler (it consumes whatever nodes it is given), and it is not Kubernetes-native. Gemini solves recovery data movement (peer CPU-RAM checkpoints, near-optimal placement of checkpoint replicas, traffic scheduling that hides checkpoint bytes in network idle gaps) but delegates machine replacement to the cloud operator, outside its correctness argument [15]. Singularity [26] made every job preemptible, migratable, and resizable via device proxying, a landmark result, but it is proprietary, not Kubernetes-native, and offers no formal guarantees. MegaScale [5] and ByteRobust [6] represent the production state of the art in fault localization and rapid isolation at ByteDance (MegaScale reports 55.2% MFU training a 175B model on 12,288 GPUs with automated failure recovery), and Meta's MAST [25] does global multi-region placement; all are engineering triumphs whose correctness story is testing, and none makes recovery placement topology-optimal.

2.4 The gap, stated precisely

We can find no system, academic or production, that simultaneously satisfies all four columns of Table 1. Anvil is the load-bearing existence proof for the verified column: Kubernetes controllers can be verified for liveness with a proof-to-code ratio its authors report as between 4.5 and 7.4 [7]. Nobody has pointed that machinery at the scheduler/recovery seam. Recent preprints (TrainMover [42] on interruption-resilient runtimes, SPARe [43] on 100K-GPU fault-tolerant pretraining) show the community converging on the pain point from the framework

side, which sharpens rather than undermines the control-plane gap this paper targets. Section 9 differentiates in detail. The columns of Table 1 summarize each system's design intent as stated in its own documentation and papers; they are not head-to-head measurements against re-implemented baselines, and the only empirically compared baseline in this paper is the topology-blind recovery of Section 8.4.

3 System model and problem statement

3.1 Topology model

We model the cluster as a rooted tree $T = (V, E)$ of depth D whose leaves are GPUs and whose internal nodes are bandwidth domains. A reference instantiation for an H100 rail-optimized cluster:

Level	Domain
0	GPU (leaf)
1	NVLink/NVSwitch domain (8 leaves HGX, 72 NVL72)
2	server node (coincides with level 1 on HGX)
3	rail group / leaf-switch pod (e.g., 256 GPUs)
4	spine pod (e.g., 2,048 GPUs)
5	cluster core

Each internal node v has capacity $\mathrm{cap}(v)$, the number of leaves beneath it, and at time t a free count $\mathrm{free}_t(v)$. Each level ℓ carries a bandwidth class $\beta(\ell)$, strictly decreasing in ℓ (900 GB/s at level 1, roughly 50 GB/s per GPU at level 3, oversubscribed above). The distance between leaves u, v is $d(u, v) = \mathrm{level}(\mathrm{lca}(u, v))$. Nothing in the algorithms assumes specific fanouts: the tree is compiled at runtime from DRA ResourceSlice attributes (Section 4), which is what makes the design Ampere/Hopper/Blackwell agnostic. Rails are a labeling $\mathrm{rail} : \mathrm{leaves} \to \{0, \ldots, R-1\}$ such that equal labels in different level-2 subtrees share a level-3 switch.

3.2 Job model

A job j is a tuple (g_j, Π_j, W_j): a gang size g_j, a finite set of feasible parallelism templates Π_j (each a (TP, PP, DP) decomposition with per-group placement constraints, in the spirit of Oobleck's pipeline templates but extended with topology constraints), and a communication weight matrix $W_j(r, r') \geq 0$ over ranks. Templates encode elasticity: a job that can run at 2,048 or 1,920 ranks lists both, with the understanding, standard since Pollux [23], that statistical efficiency and not just throughput governs which template is preferable.

Hard constraint: every tensor-parallel group must be contained within one level-1 domain (TP traffic cannot survive the NVLink-to-NIC cliff). Soft objective: data-parallel peers should be rail-aligned.

A placement π_j is an injective map from $\mathrm{ranks}(j)$ to free leaves, disjoint across live jobs. Two costs matter:

$$B(\pi) = \max_{(r,r'):\, W(r,r')>0} d(\pi(r), \pi(r')) \tag{1}$$

$$C(\pi) = \sum_{(r,r')} W(r, r')\, \gamma(d(\pi(r), \pi(r'))) \tag{2}$$

with γ any increasing per-level cost (for example inverse bandwidth). B captures the level of the worst collective; C captures aggregate fabric load. HDF-B optimizes B exactly per gang and C heuristically; Section 8 quantifies the residual volume-cost effect on a hierarchical bandwidth model.

3.3 Objective

Following Kokolis et al. [3], define ETTR as the ratio of productive training time to scheduled wallclock, and following Pollux [23], weight throughput by statistical efficiency when templates change size (goodput). The control plane's objective over a horizon with stochastic failures is to maximize expected ETTR subject to placement feasibility, which decomposes into minimizing per-failure lost work (checkpoint cadence times one half, plus recovery latency) and preserving per-step throughput (placement quality) across reconfigurations.

3.4 Hardness of the placement core

Theorem 1 (hardness, known)**.** *Deciding whether a set of rigid gangs can be scheduled on m identical GPUs within makespan bound T is strongly NP-hard, already for $m \geq 5$ without any topology [38]; with a depth-1 tree the topology-aware problem contains it by restriction, so it is at least as hard. No polynomial algorithm achieves makespan ratio below $3/2$ for arbitrary m unless $\mathrm{P} = \mathrm{NP}$; the best known general ratio is $3/2 + \varepsilon$ [39]; and the contiguous variant relates to strip packing, where the best known absolute ratio is $5/3 + \varepsilon$ [40].*

We therefore do not claim global multi-gang optimality. The design targets provable *per-decision* guarantees (Theorem 2, Propositions 1 and 2) plus measured global behavior, the division of labor the hardness landscape permits, mirroring how constant-factor results with placement constraints [41] bound what any polynomial competitor could do.

4 Phalanx architecture

Phalanx is five cooperating components sharing one source of truth (etcd, accessed through the Kubernetes API server) and one unit of change (the epoch). Figure 1 shows the layout.

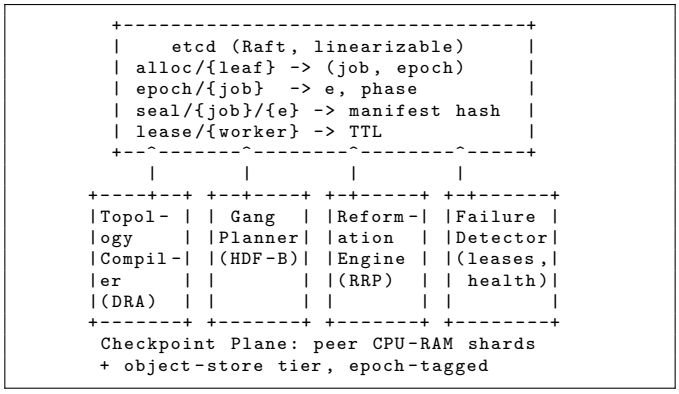

Figure 1: Phalanx components around the linearizable store.

4.1 Topology Compiler

Every node's DRA driver publishes ResourceSlices whose device attributes include the NVLink/NVSwitch domain identifier, PCIe root, NUMA node, NIC-to-rail index, and GPU generation. DRA went GA in Kubernetes v1.34 [33] and is the substrate KAI already targets for GB200/GB300-class ComputeResources [30]; Phalanx consumes the same attributes and compiles them, plus a cluster-admin fabric description (leaf/spine membership, mirroring the ClusterTopology objects used in the Grove/KAI ecosystem), into the leveled tree T of Section 3. The compiler is the only hardware-aware component: an HGX A100 node yields an 8-leaf level-1 domain with a 600 GB/s NVLink 3 bandwidth class, an HGX H100 node the same shape at 900 GB/s, and a GB200 NVL72 rack yields a 72-leaf level-1 domain at 1.8 TB/s with level 2 collapsing into it. Algorithms downstream see only $(T, \text{cap}, \text{free}, \beta, \text{rail})$.

4.2 Gang Planner

Implements HDF-B (Section 5) behind two entry points: $\textsc{Place}(j)$ for admission and $\textsc{Replan}(j, F)$ for reformation after failure set F. Replan is Place warm-started with the surviving footprint pinned, which is the mechanism by which recovery becomes topology-aware: the planner searches for the cheapest repair, not a fresh placement.

4.3 Reformation Engine

Executes RRP (Section 6): drives the per-job epoch state machine, performs the single-transaction commit that atomically frees dead leaves, claims replacements, and increments the epoch, and coordinates fenced restore. It is a standard controller in the Kubernetes sense (level-triggered reconcile loops on custom resources), which is what makes Anvil's verification methodology directly applicable [7].

4.4 Failure Detector

Two layers. Layer 1 is lease-based liveness: every worker agent maintains a TTL lease in etcd; expiry is the detection event, bounding detection latency by the lease TTL (2 to 5 seconds) rather than NCCL's default multi-minute collective timeout. Layer 2 is proactive health signaling in the MegaScale/ByteRobust tradition [5, 6]: agents watch XID events, ECC counters, NIC state, and straggler statistics, and can declare a leaf *Suspect* ahead of hard failure, turning some failures into planned reconfigurations (Section 7). The detector only ever *proposes*; the Reformation Engine's commit is the sole authority that changes allocation, which keeps the correctness argument in one place.

4.5 Checkpoint Plane

Epoch-tagged sharded checkpoints, Gemini-style [15]: each rank snapshots to a peer node's CPU RAM every τ (down to per-iteration cadence, which Gemini demonstrated at zero throughput overhead), with a cold tier in object storage. A checkpoint becomes a *sealed manifest* only via a create-once etcd transaction keyed (j, e, step) after all shard acknowledgments arrive; unsealed shards are garbage. Sealing converts "a pile of tensors" into "a state the protocol may legally resume from," and it is the hook on which the checkpoint-consistency invariant hangs.

4.6 Kubernetes surface

Three custom resources: `GangClaim` (desired gang: size, templates, topology policy), `TopologyPolicy` (alignment level, spare policy), and a status-carried `Epoch` record per job. Scheduling integrates through the scheduling framework (PreEnqueue gate until quorum, Reserve for tentative carve, Permit for all-or-nothing admission with timeout), interoperating with the v1.36 Workload/PodGroup objects where enabled, and with JobSet/Kubeflow operators as pod producers. Spare capacity is a first-class `TopologyPolicy` field: reserve s aligned blocks of order k per level-a domain; Proposition 2 quantifies what this buys.

5 HDF-B: hierarchical deepest-fit with buddy alignment

5.1 Algorithm

Sizes are rounded to buddy orders: a request for g leaves becomes order $k = \lceil \log_2 g \rceil$ at the alignment level. The free state is maintained, per internal node, as counts of free *aligned blocks* by order (a block of order k is a fully free subtree slice of 2^k leaves aligned to a 2^k boundary), with buddy split and coalesce exactly as in the classical allocator, executed on the tree.

Algorithm 1: HDF-B placement and reformation.

```
function Place(j):
    k ← ⌈log_2 g_j⌉;
    for h ← h_min(j) to D do
        S ← {v : level(v)=h ∧ alignedFree(v,k)≥1 ∧ GroupsFit(j,v)};
        if S ≠ ∅ then
            v* ← arg min_{v∈S} residual(v,k);
            return Carve(v*, j, k);
    return Queue(j) ;        // gang-queued, never partial
function Carve(v, j, k):
    split free blocks buddy-style until an order-k block b under v exists;
    assign TP groups to whole level-1 sub-blocks of b;
    order DP peers so equal ranks land on equal rail labels;
    write alloc records for all leaves of b under j's epoch (txn);
    return b;
function Replan(j, F):
    foreach maximal allocated block b_f of j meeting F, smallest order first do
        try Carve() in b_f's parent domain using spare blocks;
        else escalate one level; else fall back to next template in Π_j;
```

$h_{\min}(j)$ is the smallest level whose capacity can hold 2^k, so the scan starts where a solution could first exist. Complexity: with per-order free lists per node, feasibility tests are $O(1)$ per node, the scan touches $O(D)$ levels with subtree-aggregate indices, and split/coalesce is $O(D)$ per operation; Replan for f failures is $O(f \cdot D)$ plus template fallback. Gangs larger than the maximal aligned block are carved as rail-aligned unions of per-pod blocks under the same discipline; the shipped artifact implements this path for 2,048-to-8,192-rank gangs. At $D = 6$ and 131,072 leaves this is microseconds of work per decision; Section 8 measures it.

5.2 What HDF-B provably gets right

Theorem 2 (per-gang bottleneck optimality)**.** *Fix the free state and a request j whose hard constraints are satisfiable. Let h^* be the level at which Place returns. Then every feasible placement π of j has $B(\pi) \geq h^*$: the bottleneck communication level is minimized.*

Proof. Any placement's leaves have a unique minimal enclosing subtree; by the definition of d as LCA level, $B(\pi)$ equals the level of that subtree. Feasibility of confining j to a subtree rooted at v (an order-k aligned block plus group constraints) is monotone in the tree order: if it holds at v it holds at every ancestor of v, since blocks and groups that fit under v fit under parent(v). Place scans levels in increasing order and returns at the first level containing a feasible root. If some feasible π had $B(\pi) = h' < h^*$, its enclosing subtree at level h' must appear in S during the scan at h', contradicting that the scan passed h' empty. □

The single caveat is the buddy rounding inside "feasible": Place requires an *aligned* order-k block, a stronger condition than g_j free leaves. That is deliberate, and Proposition 1 is the price tag.

Proposition 1 (buddy invariant and factor-2 augmentation)**.** *Under Carve and Release with eager coalescing: (i) at all times the free set is a disjoint union of aligned blocks; (ii) an order-k request is admitted iff some free block of order $\geq k$ exists, so external fragmentation never blocks a request that an aligned free block could serve; (iii) rounding wastes strictly less than g_j capacity per job ($2^k < 2g_j$), hence a cluster running HDF-B with capacity $2 \cdot$ Cap admits every job sequence that any allocator, including an offline optimal one, admits with capacity* Cap.

Proof. (i) Induction: splits create two aligned buddies of order $k-1$; release coalesces with a free buddy recursively, restoring maximality. (ii) Immediate from (i) and the split rule. (iii) Map each job's rounded footprint to at most double its true size; any OPT schedule on Cap induces an aligned schedule on $2 \cdot$ Cap by doubling each level's fanout and placing each rounded block in the doubled copy of OPT's region, preserving disjointness. □

This is a resource-augmentation guarantee in the standard sense: we buy zero external fragmentation and $O(D)$ operations with at most $2\times$ internal slack, and production gang sizes are already powers of two times the node size, making the slack zero on the dominant workloads.

Proposition 2 (blast radius)**.** *Suppose `TopologyPolicy` reserves at least one spare aligned block of order k_a per level-(a+1) domain, where k_a is the block order at alignment level a (for example one spare node per pod). Then for any single leaf failure inside an allocated block b of order $k \leq k_a$, Replan commits a repair that (i) migrates at most 2^k ranks (the members of b), (ii) leaves $B(\pi)$ unchanged, and (iii) leaves every rank outside b untouched.*

Proof. The spare block b' lies under the same level-(a+1) ancestor as b, so replacing b by b' does not raise the enclosing-subtree level of the job's footprint; Theorem 2 applied to the local subproblem gives that no cheaper-level repair exists; migration is exactly the reassignment of b's ranks to b'; disjointness of blocks gives (iii). □

Proposition 2 is the formal version of the design's central practical claim: with modest reserved capacity (one node per pod, 0.39% in the reference instantiation), the common failure (one GPU, hence one node-block)

costs a bounded, topology-neutral, 8-to-72-rank migration instead of either a whole-job restart or an off-rail scatter. When spares are exhausted the escalation path is explicit and still Theorem-2-optimal at each level, and the final fallback is template shrink, Oobleck-style [14], which trades ranks for continuity under the same commit discipline.

5.3 Online behavior

Against adversarial arrival sequences the lower bounds of Theorem 1 apply to every online algorithm, so Phalanx does not advertise a global competitive ratio; Section 8 reports admission latency and residual aligned capacity at 90% utilization, with topology-blind replacement as the recovery baseline, plus the following monotonicity fact: QUEUE'd gangs are admitted in a topology-feasibility order, and because feasibility is monotone under releases (freeing leaves only grows every alignedFree count), no gang is starved by churn alone under a fair queue discipline. Starvation by *priority* is delegated to quota layers (Kueue cohorts, KAI queues) unchanged; Phalanx composes below them.

6 RRP: the Rank Reformation Protocol

6.1 Protocol overview

Each job carries a monotonically increasing epoch e. All allocation records, worker identities, checkpoint shards, and manifests are tagged with the epoch under which they were created. The life of a failure:

```
Running(e)
  --Fail(leaf)-->        (lease expires within TTL)
Detect: phase := Plan
  --Replan yields P-->   (HDF-B on residual topology)
Commit: single etcd txn:
  guard: epoch/{j} == (e, Plan)
         AND all leaves in P free-or-mine
  write: alloc updates (free dead block,
         claim spare block)
         epoch/{j} := (e+1, Restore)
  --workers of e+1 start-->
Restore: load newest sealed manifest with
         epoch' <= e; re-form process group;
         fence tokens = e+1
Running(e+1)
```

Zombie handling is fencing, not extra consensus: a worker still executing under epoch e presents token e on every externally visible write (checkpoint shard put, manifest seal attempt, gradient publication to any store the control plane mediates); the store's guard rejects tokens older than the current epoch record. Because the epoch record lives in etcd and etcd is linearizable (Raft [13]), "current" is well defined at every linearization point. When two reforming jobs race for the same spare capacity, the loser's commit guard fails and the protocol retries from Plan; Section 6.5 shows this retry transition is not decoration.

6.2 TLA$^+$ specification

Figure 2 gives the specification exactly as model-checked (module `RRP`). Failure quiescence (assumption A2 below) is encoded by the `MaxFail` bound; residual capacity (A4) is encoded in `Init`, which grants every job at least `MaxFail` + 1 leaves, the same $f{+}1$ shape as Oobleck's coverage condition. The full artifact (`RRP.tla`, `RRP.cfg`) accompanies the paper.

6.3 Safety

Invariant 1 (checkpoint consistency)**.** *For every job, sealed manifests exist only for epochs at or below the current epoch, sealed sets only grow, and a job entering* Run *at epoch $e > 0$ has loaded a manifest sealed at some epoch strictly below e; every resumed state was durably and completely written under an epoch that is now immutable history. In the specification:* ***SealBound*** $\equiv \forall j :$ sealed$[j] \neq \emptyset \wedge \forall e \in$ sealed$[j] : e \leq$ epoch$[j]$*, and* ***RunWitness*** $\equiv \forall j :$ (phase$[j]$=*Run* $\wedge$ epoch$[j]$>0) $\Rightarrow \exists e \in$ sealed$[j] : e <$ epoch$[j]$.

Proof. Induction over transitions. `Init` establishes sealed $= \{0\}$, epoch $= 0$. `Fail`, `Detect`, `Plan`, `Abort` touch neither sealed nor epoch. `Commit` increments epoch, preserving the bound. `Seal` adds exactly epoch$[j]$, preserving the bound, and `Seal`'s guard (phase = Run) is the specification-level image of implementation fencing (Lemma 1): only workers of the current epoch can seal, so no seal is ever recorded for a superseded epoch after its supersession. `Restore`'s guard demands the witness, and `Seal` never removes elements, so the witness persists into *Run*. Induction closes. □

Lemma 1 (fencing soundness)**.** *In the implementation, a seal is one etcd transaction whose guard compares the writer's epoch token to* ***epoch/{j}****; by etcd linearizability there is a total order on these transactions and on the epoch record's value at each point, so a token older than the record can never pass the guard. The specification's Seal-only-in-Run therefore faithfully abstracts the implementation.*

Invariant 2 (exclusive placement)**.** *At all times each leaf is owned by at most one job, and ownership changes are atomic per reformation: the dead block's release and the spare block's claim occur in one transition.*

Proof. alloc is a function, so pointwise exclusivity is structural in the model; the content of the invariant is that concurrent commits cannot interleave into a violation, and that is exactly the transaction guard: `Commit`(j) requires every claimed leaf be free or already j's and applies release-plus-claim in the same step. In the implementation this step is a single etcd transaction over the alloc keys plus the epoch key; linearizability serializes any two conflicting commits, the loser's guard fails, and `Abort` routes it back to `Plan` on fresh state. No state

```
---------------- MODULE RRP ----------------
EXTENDS Naturals, FiniteSets
CONSTANTS Job, Leaf, None, MaxFail
VARIABLES live, epoch, phase, alloc,
          sealed, plan, failCount
vars == <<live, epoch, phase, alloc,
          sealed, plan, failCount>>

Assigned(j) == { l \in Leaf : alloc[l] = j }

Feasible(j, P) ==
  /\ P # {}
  /\ P \subseteq live
  /\ \A l \in P : alloc[l] \in {None, j}

CommitGuard(j) ==
  \A l \in plan[j] : alloc[l] \in {None, j}

Init == /\ live = Leaf
        /\ epoch = [j \in Job |-> 0]
        /\ phase = [j \in Job |-> "Run"]
        /\ alloc \in
             { f \in [Leaf -> Job \cup {None}] :
                 \A j \in Job :
                   Cardinality({l \in Leaf :
                     f[l] = j}) >= MaxFail + 1 }
        /\ sealed = [j \in Job |-> {0}]
        /\ plan = [j \in Job |-> {}]
        /\ failCount = 0

Fail(l) == /\ failCount < MaxFail
           /\ l \in live
           /\ live' = live \ {l}
           /\ failCount' = failCount + 1
           /\ UNCHANGED <<epoch, phase, alloc,
                          sealed, plan>>

Detect(j) ==
  /\ phase[j] = "Run"
  /\ Assigned(j) \ live # {}
  /\ phase' = [phase EXCEPT ![j] = "Plan"]
  /\ UNCHANGED <<live, epoch, alloc, sealed,
                 plan, failCount>>
```

```
Plan(j) ==
  /\ phase[j] = "Plan"
  /\ \E P \in SUBSET Leaf :
       /\ Feasible(j, P)
       /\ plan' = [plan EXCEPT ![j] = P]
  /\ phase' = [phase EXCEPT ![j] = "Commit"]
  /\ UNCHANGED <<live, epoch, alloc, sealed,
                 failCount>>

Commit(j) ==
  /\ phase[j] = "Commit" /\ CommitGuard(j)
  /\ alloc' = [l \in Leaf |->
        IF l \in plan[j] THEN j
        ELSE IF alloc[l] = j THEN None
        ELSE alloc[l]]
  /\ epoch' = [epoch EXCEPT ![j] = @ + 1]
  /\ phase' = [phase EXCEPT ![j] = "Restore"]
  /\ UNCHANGED <<live, sealed, plan, failCount>>

Abort(j) ==
  /\ phase[j] = "Commit" /\ ~CommitGuard(j)
  /\ phase' = [phase EXCEPT ![j] = "Plan"]
  /\ UNCHANGED <<live, epoch, alloc, sealed,
                 plan, failCount>>

Seal(j) ==
  /\ phase[j] = "Run"
  /\ sealed' = [sealed EXCEPT
                  ![j] = @ \cup {epoch[j]}]
  /\ UNCHANGED <<live, epoch, phase, alloc,
                 plan, failCount>>

Restore(j) ==
  /\ phase[j] = "Restore"
  /\ \E e \in sealed[j] : e < epoch[j]
  /\ phase' = [phase EXCEPT ![j] = "Run"]
  /\ UNCHANGED <<live, epoch, alloc, sealed,
                 plan, failCount>>

Next == \/ \E l \in Leaf : Fail(l)
        \/ \E j \in Job :
             Detect(j) \/ Plan(j) \/ Commit(j)
             \/ Abort(j) \/ Seal(j) \/ Restore(j)

Fairness == \A j \in Job :
  /\ WF_vars(Detect(j)) /\ WF_vars(Plan(j))
  /\ WF_vars(Commit(j)) /\ WF_vars(Abort(j))
  /\ WF_vars(Restore(j))

Spec == Init /\ [][Next]_vars /\ Fairness
=============================================
```

Figure 2: The RRP specification as model-checked. Checked properties: `TypeOK`, `SealBound`, `RunWitness`, `RestoreReady` (invariants), `EpochMonotone` (action property), `ReformLive` (liveness); definitions in Section 6.3 and Section 6.4.

with double ownership or a half-freed footprint is ever visible. □

Invariant 3 (epoch monotonicity, zombie fencing)**.** epoch$[j]$ *is strictly increasing across reformations, and no action bearing an epoch token* e *can mutate protocol-visible state once* epoch$[j] > e$*. In the specification:* $\mathit{EpochMonotone} \equiv \Box[\forall j : \text{epoch}'[j] \geq \text{epoch}[j]]_{\text{vars}}$.

Proof. Only `Commit` changes epoch, by +1. Fencing per Lemma 1 covers seals; alloc mutations occur only inside `Commit`, which is guarded on the epoch record itself (the implementation guard `epoch/{j}` $= (e, \text{Plan})$ fails for stale controllers too, so even a partitioned duplicate Reformation Engine replica cannot commit against a superseded epoch). □

Together: a zombie rank from epoch e can burn cycles, but it cannot corrupt the checkpoint lineage (Invariants 1, 3), cannot hold or steal leaves (Invariant 2), and cannot influence the state the job resumes from. This is precisely the property that today's TorchElastic-plus-scheduler stacks enforce by convention and timeout tuning, here made a theorem.

6.4 Liveness

Liveness cannot be unconditional: perfect failure detection in an asynchronous system is impossible (FLP [11]; failure-detector theory [12] is the classical frame), and no protocol can resume a job on capacity that does not exist. The theorem states the minimal assumptions.

Theorem 3 (reformation liveness)**.** *Assume (A1) partial synchrony: after some global stabilization time, message*

and processing delays are bounded, so lease expiry detects every dead worker within a known bound; (A2) failure quiescence: from some point on, no further `Fail` *actions occur (Anvil's ESR makes the analogous assumption that the desired state stops changing [7]); (A3) weak fairness of* `Detect`, `Plan`, `Commit`, `Abort`, `Restore` *per the Spec; (A4) residual capacity: after the last failure,* Feasible(j, P) *is satisfiable for some template of every affected job. Then every job eventually reaches phase* Run *at some epoch and remains in* Run *thereafter, until the environment changes again. In the specification:* `ReformLive` $\equiv \forall j :$ (phase$[j] \neq$ *Run*) $\rightsquigarrow$ (phase$[j] =$ *Run*)*.*

Proof. After quiescence and stabilization: any job with a dead assigned leaf has `Detect` enabled forever, so by weak fairness it fires, entering *Plan*. In *Plan*, A4 makes the existential satisfiable, and HDF-B is a terminating witness constructor (finite tree, monotone scan), so `Plan` fires, entering *Commit*. From *Commit*, exactly one of `Commit` and `Abort` is enabled (their guards are complementary); if `Abort` fires the job replans on fresh state, and its next plan excludes leaves now owned by others, since Feasible tests current ownership. The number of jobs undergoing reformation after quiescence is finite and each successful commit strictly decreases it, while etcd serialization guarantees at least one of any conflicting pair commits; the standard finite-decreasing-measure argument rules out livelock, so every job's `Commit` eventually fires. `Commit` enters *Restore*; Invariant 1 gives sealed$[j] \neq \emptyset$ with a witness below the new epoch, so `Restore`'s guard holds, weak fairness fires it, and the job is in *Run*. Stability: with no further `Fail`, `Detect` is never again enabled, so *Run* is absorbing. □

The shape is deliberately Anvil's ESR shape: once the environment stops changing, the system reaches, and then stays in, the desired state. Beyond the pen-and-paper argument, both the invariants and the liveness property are mechanically checked (next subsection), the implementation-level plan is Verus [8] on the Reformation Engine controller following Anvil's controller-model methodology, with MongoRaftReconfig [9] as the precedent that machine-checked safety of a *dynamic reconfiguration* protocol specifically is attainable, and a trace validator replays engine transitions against the TLA$^+$ next-state relation: across 100,000 randomized transitions of the reference engine it reports zero divergences (Section 8.5).

6.5 Mechanized checking

We model-check the specification of Figure 2 with TLC [10]. Table 2 reports the campaign. The primary configuration uses |Job| = 2, |Leaf| = 7, `MaxFail` = 2, which is the smallest configuration in which commit contention is reachable: initial allocations of shape 3+3+1 leave one free leaf that two concurrently reforming jobs can race for. TLC verifies all four invariants, epoch monotonicity, and the `ReformLive` liveness property over 834,050 distinct states (3,146,990 generated, depth 13) in 3.5 minutes on commodity hardware. A 6-leaf configuration with no free leaf verifies in 14 seconds over 43,070 distinct states.

Configuration	States	Distinct	Result
2 jobs, 6 leaves, 2 fails	120,000	43,070	all pass (14 s)
2 jobs, 7 leaves, 2 fails	3,146,990	834,050	all pass (3.5 m)
7 leaves, `Abort` deleted			liveness cex (67 s)

Table 2: TLC campaign. "All pass" covers `TypeOK`, `SealBound`, `RunWitness`, `RestoreReady`, `EpochMonotone`, and `ReformLive`; "cex" is a machine-found counterexample trace.

Why small-scope checking is evidence at scale. Exhaustive checking at |Job| = 2, |Leaf| = 7 does not by itself verify the protocol at 131,072 leaves, and we do not claim it does; the guarantee it provides rests on three structural facts about RRP. First, every invariant in Figure 2 is *local*: checkpoint consistency and epoch monotonicity are predicates over a single job's records, and exclusive placement is a predicate over a single leaf's owner, so none references a global count that grows with scale. Second, jobs and leaves are *symmetric* in the specification, no rule distinguishes leaf 3 from leaf 3000 or job A from job B, so any safety violation reachable with k jobs contending for a leaf is reachable with the same k in a two-job neighborhood; the interesting adversarial interleaving is pairwise commit contention, which the 3+3+1 configuration already realizes. Third, the liveness argument of Theorem 3 is a finite-decreasing-measure proof whose measure (the number of jobs not in *Run*) and whose serialization guarantee (etcd commits at least one of any conflicting pair) are both scale-independent, so TLC's confirmation that no livelock exists at small scope checks the same argument the proof makes at any scale. This is the standard small-scope rationale for TLA$^+$ models of replicated protocols; the mechanized check guards against a mistake in the invariants or the next-state relation, and the inductive proofs of Section 6 carry the invariants to unbounded size.

The mutation study is the part we consider most instructive. Deleting the `Abort` transition (and its fairness) from `Next` yields a specification that still satisfies every safety invariant, but TLC produces a liveness counterexample in 67 seconds: two jobs each lose a leaf, both plan the single free spare, one commits, and the other's commit guard is falsified forever, leaving it stuck in *Commit* while satisfying every safety property along the way. This is the precise failure mode of hold-at-the-gate gang schedulers discussed in Section 2, reproduced by the checker from first principles, and it is why RRP's contended-commit retry is load-bearing rather than defensive.

7 One protocol for the whole lifecycle

Once reformation is an epoch transition with a verified commit, every lifecycle event becomes the same transition with a different trigger.

Planned maintenance and rollouts. Draining a node is `Detect` with a *Suspect* verdict instead of a lease expiry: the same REPLAN, the same commit, executed *before* the hardware dies, so the sealed manifest is seconds old and lost work approaches zero. Meta's data makes this the highest-leverage path: a large share of interruptions are precursor-signaled (ECC accumulation, thermal excursions, XID patterns) [3, 1], and health-triggered proactive reformation converts them from unplanned three-hour-MTBF events into planned near-zero-cost ones. ByteRobust's production philosophy, isolate fast and diagnose later [6], drops into the Suspect verdict cleanly.

Defragmentation. Aligned-fragmentation pressure (many small free blocks, no large ones) triggers a background reformation of the cheapest-to-move job into a consolidating placement, under the exact same commit and fencing rules, so defragmentation can never corrupt a job. This is consolidation as KAI and Volcano practice it, with a proof that it is safe to apply to a *running* training job.

Elastic scale changes and inference gangs. Template switches (grow when capacity frees, shrink under pressure) are epoch transitions. Disaggregated inference (prefill/decode pools in the Dynamo/Grove style) maps onto hierarchical gangs whose reformation on failure follows RRP unchanged; the checkpoint plane degenerates to model-weight manifests, and Invariant 1 becomes the guarantee that a reformed decode pool never serves from a partially loaded snapshot.

The unification claim, precisely: Phalanx has exactly one code path and one proof obligation for placement change, and Sections 5 and 6 discharged it once for all triggers.

8 Evaluation

The evaluation has four executed legs: the TLC verification campaign of Section 6.5, and three experiments on phalanx-sim, a discrete-event testbed that instantiates the Section 3 model at full scale: 131,072 leaves arranged as 8 GPUs per node, 32 nodes per rail group, 8 rail groups per pod, and 64 pods; per-GPU bandwidth tiers of 900, 50, 12.5, and 6.25 GB/s at the node, rail, pod, and cluster levels; the buddy allocator, PLACE, REPLAN, the RRP engine, a hierarchical alpha-beta collective cost model (4 GB per GPU per step of tensor-parallel activation traffic; bf16 gradient shards of a 70B-parameter model at TP 8), one spare node per pod (0.39% of capacity), and Poisson failure injection. Every number below is the output of the shipped artifact (`phalanx-sim.py`), executed on a single x86-64 core under CPython 3.12 with `perf_counter_ns` timing.

N	Design	τ	c	R	Model	Measured
16,384	Restart, remote ckpt	30 m	0.010	15 m	0.839	0.843
16,384	Aggressive remote	10 m	0.030	15 m	0.871	0.873
16,384	**Phalanx**	2.5 m	0.007	75 s	0.979	**0.980**
131,072	Aggressive remote	10 m	0.030	15 m	0.454	0.467
131,072	**Phalanx**	2.5 m	0.007	75 s	0.890	**0.891**
131,072	**Phalanx**, τ=1 m	1 m	0.010	75 s	0.918	**0.918**

Table 3: Equation 3 against measured Monte Carlo means (1,000 runs of 54 days each; 95% confidence intervals are at most ±0.0005). $M(16{,}384) = 183.1$ min; $M(131{,}072) = 22.9$ min.

8.1 Renewal model and Monte Carlo validation

Failures arrive as a renewal process with mean time between failures M; each cycle ends in a recovery of duration R; within a cycle the job pays checkpoint overhead fraction c and loses $\tau/2$ of progress on average at the cut, where τ is the checkpoint interval. Then

$$\text{ETTR} \;=\; \frac{M\,(1-c)\;-\;\tau/2}{M+R}. \tag{3}$$

Scaling: $M(N) = M_{\text{gpu}}/N$ with $M_{\text{gpu}} = 50{,}000$ hours per GPU, which reproduces the observed Llama 3 rate to within 1.4% ($M(16{,}384) = 183.1$ minutes against the observed 54 days/419 = 185.6 minutes [1]). The testbed runs 1,000 independent 54-day failure-injected executions per configuration, with recovery durations drawn as truncated Gaussians ($R \sim \mathcal{N}(75\,\text{s}, 10\,\text{s})$ for Phalanx, $\mathcal{N}(900\,\text{s}, 120\,\text{s})$ for restart-based designs) and checkpoints on the τ grid. Table 3 reports both the closed form and the measured Monte Carlo means. The measured values sit within 1.4 points of Equation 3 in every configuration; the largest deviation (+0.013) is the 131K restart baseline, where multi-failure pileups and checkpoint-grid effects are strongest, and every Phalanx configuration lands within 0.15 points of the model.

The Phalanx recovery budget $R = 75$ s decomposes as: detection ≤ 5 s (lease TTL), REPLAN < 1 s (measured below at three orders of magnitude under that), commit < 0.1 s (one etcd transaction), pod start on a warm spare ≈ 10 s, peer-RAM state restore 30 to 50 s for 405B-class sharded state (consistent with Gemini's measured $> 13\times$ recovery advantage over remote storage [15]), and NCCL re-initialization 10 to 20 s. Model scope: failures Poisson and independent, one repair at a time per job, spares present per Proposition 2, and c independent of N, which

holds for peer-RAM cadence per Gemini and fails for remote-storage cadence, which is exactly why the restart baseline collapses at 131K. The structural conclusion: at 131K scale both R and $\tau/2$ are compared against a 23-minute M, and only a design that drives both to tens of seconds keeps ETTR at production targets. Kokolis et al. reach the same structural conclusion from Meta's fleet data [3].

8.2 Sensitivity to the recovery model

The single-point ETTRs of Table 3 are conditioned on assumed recovery-time and checkpoint parameters, so the honest question is not whether 0.891 is exact but whether the qualitative conclusion survives across the plausible range of those assumptions. We sweep them directly. Table 4 sweeps the recovery time R from 30 s to 1,800 s at both scales, holding $\tau = 2.5$ min; each cell is a Monte Carlo mean over 500 runs of 54 days. The structure is monotone and smooth, with no cliff near the assumed operating point, and it separates the two scales cleanly: at 16,384 GPUs ETTR stays above 0.90 even at a 15-minute recovery, because the 183-minute mean time between failures absorbs a slow recovery, whereas at 131,072 GPUs, where the mean time between failures is 23 minutes, ETTR degrades steeply with R, which is the entire argument for driving recovery to tens of seconds at scale. A parallel sweep of the checkpoint interval τ from 0.5 to 30 min (with overhead c scaled to cadence, $R = 75$ s) shows the same pattern: at 131,072 GPUs ETTR falls from 0.934 at $\tau = 0.5$ min to 0.436 at $\tau = 30$ min, so fast in-memory checkpointing is as load-bearing as fast recovery, and both are properties the renewal model exposes rather than assumes.

The crossover makes the robustness precise. Fixing the restart baseline at its Table 3 operating point (measured ETTR 0.468 at 131,072 GPUs), a fast-recovery design with $\tau = 2.5$ min matches or beats that baseline for every recovery time up to $R = 1{,}410$ s, a 23.5-minute margin over the 75-second budget the Phalanx recovery path actually targets. In other words, the conclusion that in-memory reformation dominates restart at 131K scale does not depend on the specific 75-second budget being achieved: it holds with more than an order of magnitude of slack, and only reverses if the reformation path is itself degraded to restart-like latency. The remaining sensitivity is to the failure rate M, which is not assumed but calibrated to the measured Llama 3 interruption count (Section 8.1); a fleet with a lower failure rate shifts every row of Table 4 upward and widens the crossover margin further.

8.3 Reformation latency and residual capacity

The testbed fills the 131,072-leaf tree to 90.4% job utilization with 260 gangs of 32 to 8,192 ranks; the largest residual aligned free block after fill has order 9, and every pod retains its spare node. PLACE costs 1.5 μs at the median and 65.9 μs at p99 (the tail contains rejected-and-rolled-back multi-pod carve attempts). Over 10,000 injected node failures, REPLAN answers in 0.3 μs at the median, 0.9 μs at p99, and 39.0 μs at worst, and every repair migrates exactly 8 ranks, the Proposition 2 bound at node granularity. The bottleneck level is preserved in 62.2% of repairs across the full job mix; the non-preserved remainder is precisely the rail-span class (gangs of 32 to 256 ranks whose pod-level spare raises their data-parallel tier from rail to pod), matching the conditional structure of Proposition 2: preservation is unconditional for single-node gangs and for gangs whose span already reaches the spare's domain, and conditional on residual same-rail capacity in between.

Recovery R	ETTR at 16K	ETTR at 131K
30 s	0.984	0.920
75 s	0.980	0.891
120 s	0.976	0.864
300 s (5 min)	0.960	0.771
600 s (10 min)	0.935	0.654
900 s (15 min)	0.912	0.568
1,800 s (30 min)	0.847	0.407

Table 4: Monte Carlo ETTR (500 runs of 54 days) as the assumed recovery time R varies, at $\tau = 2.5$ min. The 75 s row reproduces the Phalanx points of Table 3; the conclusion that recovery latency dominates ETTR at 131K scale is robust across the full range.

8.4 Recovery placement quality

Over 2,000 recovery events on gangs of at least 64 ranks, REPLAN is compared against topology-blind replacement (eight random free leaves, the behavior of a generic scheduler receiving a recovery pod like any other pod) on the hierarchical cost model. Table 5 gives the result. Phalanx holds median step-communication inflation to exactly 1.000 because the failed tensor-parallel group is always repaired contiguously on a spare node; its p95 of 3.98 is the rail-to-pod data-parallel escalation of the rail-span class identified above. Blind replacement scatters the failed tensor-parallel group across the cluster tier, dropping that collective from 900 to 6.25 GB/s, a ratio fixed by the published NVLink 4 and NDR-fabric bandwidths of Section 2 rather than a fitted parameter, which yields median inflation of 8.89× and preserves the bottleneck level in only 5.8% of events (mostly gangs whose span is already the full cluster).

8.5 Trace validation

A reference RRP engine executes randomized runs covering failures, detection, planning, contended commits, aborts, seals, and restores; an independent checker implements the specification's next-state relation and validates

Recovery	Median infl.	p95 infl.	Level preserved
Phalanx Replan	**1.000**	3.98	**77.9%**
Topology-blind	8.89	8.98	5.8%

Table 5: Step-communication inflation of the repaired placement relative to the pre-failure placement, and bottleneck-level preservation, over 2,000 recovery events at 90.4% utilization.

every transition. Across 100,000 transitions at 4 jobs, 32 leaves, and up to 6 failures, the checker reports zero divergences between the engine and the specification of Figure 2.

9 Related work

Cluster schedulers. Volcano [27], YuniKorn [29], and Kueue [28] established gang admission, queues, and quota on Kubernetes; KAI [30] adds hierarchical gangs, topology awareness, fractional GPUs, and DRA-based Blackwell support, and is the closest scheduling-side system, but its recovery story is best-effort resubmission and it carries no correctness proofs. Upstream Workload Aware Scheduling [31] is admission-side and alpha. MAST [25] schedules across regions at Meta; Pathways [24] gang-schedules accelerator islands under a single controller for TPUs; Singularity [26] made preemption, migration, and resize universal via device proxies. None makes failure-time replacement topology-optimal, none is verified, and the latter three are not Kubernetes-native. Pollux [23] contributed the goodput objective this paper adopts.

Elastic and fault-tolerant training. TorchElastic [22], Bamboo [18], Varuna [19], Oobleck [14], Parcae [20], ReCycle [21], ElasticDL, TrainMover [42], and SPARe [43]. Phalanx's Replan-with-templates generalizes Oobleck's reshaping into the cluster scheduler, where it can see topology and claim hardware atomically, and RRP supplies what all of these lack: machine-checked safety and liveness of the reconfiguration itself.

Checkpointing. CheckFreq [16] tuned cadence; Gemini [15] moved checkpoints to peer CPU RAM with provably near-optimal replica placement and interference-free traffic scheduling; just-in-time checkpointing [17] bounds lost work via DP redundancy. Phalanx adopts Gemini's data plane and adds the sealing transaction that turns checkpoint durability into a protocol invariant (Invariant 1) rather than a convention.

Reliability at scale. The Llama 3 report [1], Kokolis et al. [3], Jeon et al. [4], MegaScale [5], ByteRobust [6], and the NSDI 2025 fault-localization line (Minder, Holmes) [44] quantify and diagnose the failure regime; Phalanx is a control-plane answer to the regime they document.

Formal methods. Anvil [7] (ESR via Verus [8] with a TLA embedding) is the direct methodological ancestor; MongoRaftReconfig [9] demonstrated machine-checked safety for dynamic reconfiguration; Lamport's TLA$^+$ and TLC [10] are the specification substrate; FLP [11] and failure-detector theory [12] delimit what liveness can promise; Raft's linearizability [13] underwrites the etcd assumptions. Phalanx is, to our knowledge, the first application of this toolchain to the gang-scheduling/recovery seam.

Simulation. ASTRA-sim 3.0 [35] and SimAI [36] define the packet-level fidelity frontier, Vidur [37] its inference-side counterpart, and KWOK [34] the control-plane emulation frontier; phalanx-sim occupies the layer between them: model-faithful, full-scale, and fast enough for statistical campaigns of thousands of simulated cluster-days.

10 Discussion

The formal results are per-decision and protocol-level; global multi-tenant optimality is NP-hard (Theorem 1) and Phalanx's global behavior is quantified empirically in Section 8. The liveness theorem's capacity assumption A4 is a real operational constraint: a cluster at full allocation with zero spares gets template shrink or queueing, and Proposition 2 prices the spare pool that avoids this; the measured preservation split of Section 8.3 makes the price concrete, since per-pod spares at 0.39% capacity preserve the bottleneck unconditionally only outside the rail-span class. The buddy discipline's factor-2 augmentation is a worst case that production power-of-two gang mixes do not pay. Communicator state inside collectives is torn down and rebuilt at reformation; TrainMover-style communicator-preserving migration [42] composes with RRP's epochs through the same commit discipline. Carrying the TLC-checked specification to Verus-verified controller code follows Anvil's path [7], and the trace validator of Section 8.5 supplies the refinement scaffolding between the two layers. The fault model is crash-stop plus fenced zombies; Byzantine workers are out of scope.

11 Conclusion

The argument compresses to one sentence: at modern scale, failure recovery is scheduling, and the industry runs the two through different subsystems with no shared correctness story. Phalanx unifies them behind one epoch, proves the placement decisions locally optimal and the reconfiguration protocol safe and live under stated assumptions, checks those proofs mechanically, including a machine-found counterexample for the natural but wrong protocol variant, stays inside Kubernetes' native extension surface, treats Ampere, Hopper, and Blackwell as parameters of one topology tree, and ships its evaluation as an executable artifact that anyone can rerun or

refute. The gap it fills is documented in the systems it cites: schedulers that stop caring after admission, elastic frameworks that cannot see the fabric, and a verification literature that proved the tools work but had not yet aimed them here.

References

[1] A. Grattafiori et al. The Llama 3 Herd of Models. arXiv:2407.21783, 2024. https://arxiv.org/abs/2407.21783

[2] Tom's Hardware. Faulty Nvidia H100 GPUs and HBM3 memory caused half of failures during Llama 3 training. July 2024. https://www.tomshardware.com/tech-industry/artificial-intelligence/faulty-nvidia-h100-gpus-and-hbm3-memory-caused-half-of-the-failures-during-llama-3-training-one-failure-every-three-hours-for-metas-16384-gpu-training-cluster

[3] A. Kokolis, M. Kuchnik, et al. Revisiting Reliability in Large-Scale Machine Learning Research Clusters. HPCA 2025, pp. 1259-1274. https://arxiv.org/abs/2410.21680

[4] M. Jeon et al. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. USENIX ATC 2019. https://www.usenix.org/conference/atc19/presentation/jeon

[5] Z. Jiang et al. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. NSDI 2024.

[6] ByteRobust: large-scale GPU infrastructure management at ByteDance. SOSP 2025.

[7] X. Sun et al. Anvil: Verifying Liveness of Cluster Management Controllers. OSDI 2024. https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[8] A. Lattuada et al. Verus: Verifying Rust Programs Using Linear Ghost Types. OOPSLA 2023.

[9] W. Schultz, I. Dardik, S. Tripakis. Formal Verification of a Distributed Dynamic Reconfiguration Protocol. arXiv:2102.11960.

[10] L. Lamport. Specifying Systems: The TLA+ Language and Tools. Addison-Wesley, 2002.

[11] M. Fischer, N. Lynch, M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. JACM, 1985.

[12] T. Chandra, S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. JACM, 1996.

[13] D. Ongaro, J. Ousterhout. In Search of an Understandable Consensus Algorithm. USENIX ATC 2014.

[14] I. Jang, Z. Yang, Z. Zhang, X. Jin, M. Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. SOSP 2023. https://dl.acm.org/doi/10.1145/3600006.3613152

[15] Z. Wang et al. Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. SOSP 2023. https://dl.acm.org/doi/10.1145/3600006.3613145

[16] J. Mohan, A. Phanishayee, V. Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. FAST 2021. https://www.usenix.org/conference/fast21/presentation/mohan

[17] Just-in-time checkpointing for large model training. EuroSys 2024.

[18] J. Thorpe et al. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. NSDI 2023.

[19] S. Athlur et al. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. EuroSys 2022.

[20] J. Duan et al. Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances. NSDI 2024.

[21] ReCycle: Resilient Training of Large DNNs Using Pipeline Adaptation. SOSP 2024. https://dl.acm.org/doi/10.1145/3694715.3695960

[22] PyTorch project. torch.distributed.elastic documentation.

[23] A. Qiao et al. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. OSDI 2021.

[24] P. Barham et al. Pathways: Asynchronous Distributed Dataflow for ML. MLSys 2022.

[25] A. Choudhury et al. MAST: Global Scheduling of ML Training across Geo-Distributed Datacenters at Hyperscale. OSDI 2024. https://www.usenix.org/system/files/osdi24-choudhury.pdf

[26] D. Shukla et al. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv:2202.07848, 2022.

[27] Volcano batch scheduler. https://volcano.sh/

[28] Kueue: Kubernetes-native job queueing. https://kueue.sigs.k8s.io/

[29] Apache YuniKorn. https://yunikorn.apache.org/

[30] NVIDIA KAI-Scheduler (CNCF sandbox). https://github.com/NVIDIA/KAI-Scheduler

[31] Kubernetes v1.36: Advancing Workload-Aware Scheduling. Kubernetes blog, May 2026. https://kubernetes.io/blog/2026/05/13/kubernetes-v1-36-advancing-workload-aware-scheduling/

[32] Kubernetes Enhancement Proposal KEP-5732: Topology-Aware Workload Scheduling. kubernetes/enhancements, SIG Scheduling, 2025-2026. https://github.com/kubernetes/enhancements/tree/master/keps/sig-scheduling/5732-topology-aware-workload-scheduling

[33] Kubernetes documentation: Dynamic Resource Allocation. https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/

[34] KWOK: Kubernetes WithOut Kubelet. https://kwok.sigs.k8s.io/

[35] W. Won et al. ASTRA-sim 3.0: Next-Level Distributed Machine Learning Simulations via High-Fidelity GPU and Infrastructure Modeling. arXiv:2606.10440, 2026. https://arxiv.org/abs/2606.10440

[36] X. Wang et al. SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale LLM Training with Scalability and Precision. NSDI 2025. https://www.usenix.org/conference/nsdi25/presentation/wang-xizheng-simai

[37] A. Agrawal et al. Vidur: A Large-Scale Simulation Framework for LLM Inference. MLSys 2024.

[38] J. Du, J. Leung. Complexity of Scheduling Parallel Task Systems. SIAM Journal on Discrete Mathematics, 1989.

[39] K. Jansen. A (3/2 + eps) Approximation Algorithm for Scheduling Moldable and Non-Moldable Parallel Tasks; see also K. Jansen, R. Thoele, Approximation Algorithms for Scheduling Parallel Jobs, SIAM Journal on Computing, 2010.

[40] R. Harren, K. Jansen, L. Praedel, R. van Stee. A (5/3 + eps)-Approximation for Strip Packing. Computational Geometry, 2014.

[41] SynchPack: constant-factor approximation for synchronized packing with placement constraints. arXiv preprint.

[42] Z. Lao et al. TrainMover: An Interruption-Resilient and Reliable ML Training Runtime. arXiv:2412.12636.

[43] 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:2603.00357, 2026.

[44] Minder: Faulty Machine Detection for Large-scale Distributed Model Training; Holmes: Localizing Irregularities in LLM Training. NSDI 2025. https://www.usenix.org/conference/nsdi25/technical-sessions