HELIOS: Failure-Aware Distributed Scheduling for Heterogeneous AI Services

Mohammed Kamran Syed[1], Anudeep Rentala[2], Somesh Rahul[2], Raghuram Malpe Pai[3], Krishna Suman Dara[3]

IEEE Independent Researcher, Mountain House, CA, USA[1], IEEE Independent Researcher, Fremont, CA, USA[2], IEEE Independent Researcher, Sunnyvale, CA, USA[3]

Corresponding author:Mohammed Kamran Syed, Email: mohammedkamran.syed@ieee.org

Distributed AI services now run across mixed GPU generations, CPU pools, memory tiers, network domains, caches, replicas, and geographically separated clusters. Most cluster schedulers still optimize for instantaneous load, locality, or fairness, so they cope poorly when a workload burst, a degrading network link, and a component failure all hit at once. HELIOS is a failure-aware distributed scheduler that ties placement, locality, replication, migration, and recovery into a single decision loop instead of running them as separate reactive mechanisms. It keeps a resource and failure-risk graph up to date, forecasts short-horizon saturation, scores candidate placements on latency, locality, cost, and resilience, and moves state, through replication or live migration, before an overload becomes visible to users. On heterogeneous AI workloads, HELIOS cuts P95 latency by 35.8% and P99 by 38.3%, lifts throughput 1.34×, pushes GPU utilization to 84.6%, and cuts SLA violations by 62.8% against round-robin scheduling. Under injected failures it recovers successfully 93.7% of the time on average and cuts weighted MTTR by 53.6%. It does this for 5.4 ms of mean scheduling overhead and 18.9% lower operating cost overall. The takeaway is that performance, placement, and recovery work better as one coordinated control problem than as three independent ones.

Keywords: Distributed systems, cluster scheduling, heterogeneous computing, fault tolerance, GPU scheduling, predictive placement, live migration, replication, AI infrastructure.

2026. In Mukesh Saraswat & Harish Sharma (eds.), *Emerging Technologies and Intelligent Systems: Theory, Applications and Innovations*, 120–141. Computing & Intelligent Systems, SCRS, India. DOI: https://doi.org/10.56155/978-81-975670-4-9-7

1 Introduction

Scheduling for distributed AI systems is harder than it looks. A single service might combine latency-sensitive inference, memory-heavy model state, bursty arrivals, accelerator-specific kernels, distributed caches, and replicated control-plane state. The cluster underneath is just as uneven, with workers differing in GPU generation, memory, network locality, failure history, energy efficiency, and model warmth. A placement decision that looks optimal at admission time can be wrong seconds later: queue pressure rises, a link degrades, a replica goes unhealthy, or a correlated failure removes capacity outright.

Classical cluster schedulers already gave us strong abstractions for sharing, fairness, and locality: Borg [1], Omega [2], Mesos [4], Sparrow [5], Quincy [6], and Firmament [8] all show what centralized or logically centralized coordination buys at scale. More recent AI schedulers such as Gandiva [10], Tiresias [12], Optimus [13], Gavel [14], and Pollux [16] layer on GPU-aware allocation and job-level adaptation, and serving systems like Clipper [18], InferLine [20], Clockwork [21], vLLM [22], AlpaServe [24], DistServe [25], Splitwise [26], and Sarathi-Serve [28] push further with batching, memory management, phase separation, and queue-aware execution. In practice, though, these pieces tend to run as separate loops: a placement scheduler picks a worker, a load balancer reacts to queue buildup, a replication policy guards availability on its own schedule, and a recovery subsystem only wakes up once something has already failed.

That separation gets expensive fast. Take a GPU worker whose queue is climbing while its network path is quietly degrading: a load-only scheduler keeps placing work there because compute is technically free, a locality-only scheduler keeps favoring it for data-near placement, and a failure-only mechanism does nothing because the worker hasn't failed yet. By the time any of them reacts, tail latency has already spiked and state has to move under pressure, not ahead of it. The better call was available the whole time: cut new admissions to that worker, prewarm a replica, move some state out early. It just wasn't available to any single mechanism looking at only its own signal.

Our argument is that placement, load balancing, replication, migration, and recovery should work off one shared, predictive view of cluster state rather than four or five separate ones. HELIOS builds that view as a resource graph covering worker capacity, network paths, state locality, replicas, and failure risk, and refreshes it every scheduling epoch. From there it forecasts short-horizon load and risk, generates feasible placements, and picks among them by minimizing a multi-objective score under capacity and availability constraints. It replicates

or migrates state proactively whenever the expected cost of doing nothing outweighs the cost of moving, and when a failure does happen, recovery runs against the same graph, without handing off to a separate, disconnected failover path.

Related work has explored complementary mechanisms in distributed LLM resource allocation [3], edge–cloud partitioning [7], multi-tier load balancing [11], adaptive token routing [15], warm-start inference [19], multi-tenant placement [23], monitoring [27], self-healing [31], root-cause analysis [35], and adaptive GPU resource management [38]. HELIOS differs by pulling these threads together into one cluster-wide scheduling problem, with replication and migration treated as explicit decisions, not side effects.

The main contributions are:

- **Unified failure-aware scheduling model.** HELIOS jointly optimizes worker placement, locality, load, failure exposure, replication state, and migration cost together, in one score, instead of bolting recovery on after the fact.
- **Predictive resource and risk graph.** The scheduler keeps a short-horizon forecast of GPU load, queue saturation, memory pressure, network degradation, and failure risk, and feeds it directly into candidate generation.
- **Risk-aware replication and migration.** HELIOS prewarms replicas and migrates state selectively, only when the predicted drop in tail latency or failure exposure is worth more than the move itself costs.
- **Graph-preserving recovery.** Failures simply remove or penalize resources in the same placement graph, so recovery reuses the normal scheduler without relying on special-case failover logic.
- **Comprehensive distributed evaluation.** We measure tail latency, throughput, utilization, load imbalance, recovery success, MTTR, migration overhead, scalability, burst tolerance, network sensitivity, replication efficiency, and operating cost.

2 Design Motivation

2.1 Why Reactive Load Balancing Is Insufficient

Least-loaded and queue-aware policies beat round robin, but they're still fundamentally reactive: they only respond once load is already visible, they ignore what moving state will cost later, and they can oscillate when several schedulers

chase the same hotspot at once. For AI services the cost of a late move is especially high: model state and KV caches are often large, accelerator compatibility narrows where a job can even go, and a cold worker takes a while to warm up. That combination is why HELIOS leans on prediction and hysteresis instead of ranking workers by instantaneous load alone.

2.2 Why Failure Recovery Must Influence Placement

A placement's real cost isn't just its normal-case latency. A worker can look great on throughput today and still carry a high expected recovery cost because its network redundancy is thin or its health signals have been erratic. The opposite failure mode, replicating everything everywhere, just burns scarce GPU memory. HELIOS treats resilience as something to optimize continuously, not react to: failure exposure factors into every normal placement decision, and replication gets allocated by workload criticality and predicted risk instead of a flat policy.

2.3 Design Goals

HELIOS targets six properties: (G1) low tail latency, (G2) high accelerator utilization, (G3) low load imbalance, (G4) fast and successful recovery, (G5) bounded migration/control overhead, and (G6) scalable scheduling across heterogeneous nodes.

3 System Model

3.1 Cluster State

We model the cluster as a directed graph $G_t = (V, E_t)$ at time t. A worker $i \in V$ has state

$$x_i(t) = [g_i, c_i, m_i, q_i, w_i, h_i, r_i]. \tag{7.1}$$

where g_i is accelerator capacity, c_i CPU capacity, m_i available memory, q_i queue state, w_i model/cache warmth, h_i health score, and r_i predicted failure risk. An edge $(i, j) \in E_t$ carries network state

$$e_{ij}(t) = [b_{ij}, d_{ij}, p_{ij}]. \tag{7.2}$$

with bandwidth b, latency d, and degradation probability p. A request or service shard k has a demand vector

$$d_k = [g_k, c_k, m_k, n_k, s_k, \rho_k]. \tag{7.3}$$

where n_k is network demand, s_k movable state size, and ρ_k is the service availability tier.

3.2 Placement Objective

For feasible worker i, HELIOS predicts latency $\hat{L}_{ki}$, queue penalty $\widehat{Q}_{ki}$, locality penalty D_{ki}, monetary cost C_{ki}, migration cost M_{ki}, and failure exposure F_{ki}. The placement score is

$$J(k,i) = \alpha\hat{L}_{ki} + \beta\widehat{Q}_{ki} + \gamma D_{ki} + \delta C_{ki} + \eta M_{ki} + \lambda F_{ki}. \tag{7.4}$$

The weights are service-class specific. Feasibility requires capacity, hardware compatibility, region policy, and replica constraints.

3.3 Replication and Migration

Let R_k be the current replica set. HELIOS creates an additional replica when expected risk-adjusted loss exceeds replica cost:

$$\Pr[\text{loss} \mid R_k] P_k > C_k^{\text{rep}} + C_k^{\text{mem}}, \tag{7.5}$$

where P_k is the service penalty of unavailability. A migration from i to j is selected when the expected reduction in future placement cost exceeds transfer and disruption cost by hysteresis margin ϵ:

$$\mathbb{E}[J_{ki}^H] - \mathbb{E}[J_{kj}^H] > M_{ij} + \epsilon. \tag{7.6}$$

This prevents oscillation under short-lived load changes.

3.4 Complexity and Stability

Each epoch performs two graph-scale passes and one per-shard pass. AnnotateRisk updates load and failure-risk estimates over the full graph, costing $O(|V| + |E|)$; FeasibleWorkers then prunes each shard's candidate set by hardware, memory, and region constraints before scoring, so the number of workers scored per shard is bounded by a small constant c independent of $|V|$ in a well-partitioned cluster. Per-epoch cost is therefore $O(|V| + |E| + |K|c)$, dominated by

the graph pass rather than by per-shard scoring. This matches the scaling in Table 6: control latency grows only 2.5× (2.8 to 7.1 ms) as the cluster grows 32× (8 to 256 nodes), consistent with a cost that scales with graph size rather than with the product $|K||V|$.

The hysteresis margin ϵ in Eq. (7.6) also gives a simple guarantee against migration thrashing. Suppose predicted placement scores drift by at most δ within one decision horizon H, a reasonable assumption given that the prediction layer in Section 4.1 is calibrated over the same horizon. If $\epsilon > 2\delta$, no shard migrates more than once per horizon: a migration from i to j requires $\mathbb{E}[J_{ki}^{H}] - \mathbb{E}[J_{kj}^{H}] > M_{ij} + \epsilon$, and a subsequent reverse migration from j back to i within the same horizon would require the score gap to swing by more than 2ϵ in the opposite direction, which exceeds what δ-bounded drift can produce once $\epsilon > 2\delta$. The bound is conservative: in practice δ is measured directly from rolling forecast error and ϵ is set above it with margin. But it explains why HELIOS does not need an explicit cooldown timer to avoid oscillation.

The replication rule in Eq. (7.5) is not a heuristic threshold; it is the minimizer of expected cost under a linear loss model. Treating the replicate/do-not-replicate choice as a binary decision $z \in \{0, 1\}$, expected cost is $\mathbb{E}[C] = (1 - z)\Pr[\text{loss} \mid R_k]P_k + z(C_k^{\text{rep}} + C_k^{\text{mem}})$. Because the two terms are linear in z, the cost-minimizing choice is $z = 1$ exactly when $\Pr[\text{loss} \mid R_k]P_k$ exceeds $C_k^{\text{rep}} + C_k^{\text{mem}}$, precisely the condition HELIOS evaluates. The same argument extends to the migration rule in Eq. (7.6), with M_{ij} playing the role of the fixed cost and the placement-score gap playing the role of the avoided loss.

4 HELIOS Architecture

Figure 1 lays out the control plane. Telemetry flows into a predictive state engine, a graph builder folds in capacity, topology, locality, and health, the scheduler scores the resulting feasible placements, and local agents carry out whatever replication or migration actions get selected. A failure event updates that same graph and kicks off rescheduling immediately, without a separate path.

4.1 Predictive State Engine

The prediction layer forecasts GPU saturation, queue length, memory pressure, network degradation, and failure risk a short horizon ahead, using deliberately lightweight models, since prediction has to stay cheaper than scheduling itself, not compete with it. Forecasts are calibrated against rolling telemetry and treated

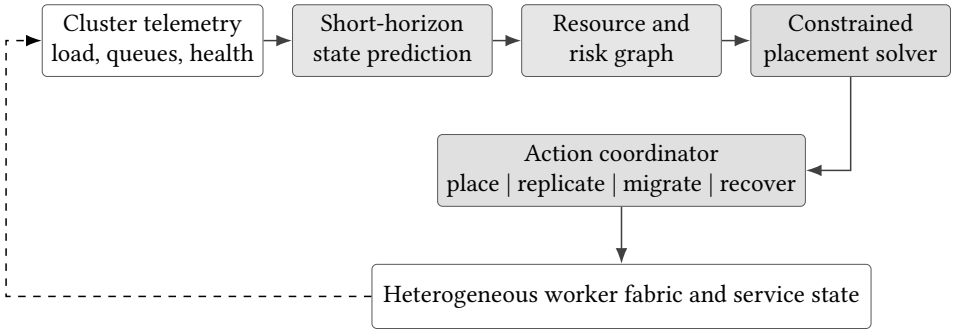


Figure 1: HELIOS control plane. Telemetry is converted into predicted cluster state and a shared resource/risk graph. The scheduler selects a feasible plan, while an explicit action coordinator independently commits placement, replication, migration, and recovery actions. Runtime telemetry and health events close the control loop without crossing the primary decision path.

as soft evidence, never as ground truth; capacity and policy checks stay strictly deterministic underneath them.

4.2 Candidate Generation

Candidate workers get filtered first by accelerator compatibility, memory capacity, region constraints, and whatever state the job already needs, and HELIOS then widens that set to include migration destinations and replica targets. Locality-aware pruning favors workers that already hold the relevant model or cache state; risk-aware pruning drops any destination whose predicted failure exposure is too high for the service's tier.

4.3 Hysteretic Placement and Migration

To keep the scheduler from thrashing, a placement advantage has to either persist for a minimum decision horizon or clear a larger emergency threshold before HELIOS acts on it. Migration itself moves in chunks, prioritizes warm state, and keeps the source replica alive until the destination's health is confirmed, so state can move ahead of trouble without availability taking the hit.

4.4 Recovery Semantics

When a failure hits, the affected worker or edge is removed or heavily penalized in G_t, surviving replicas get promoted, requests reroute, and any missing redun-

dancy is rebuilt once the service has stabilized. This graph-preserving approach draws on self-healing methods [31] and monitoring and diagnosis work [27, 35], applied here at the scale of cluster scheduling rather than a single service.

5 Scheduling Algorithm

Algorithm 1 summarizes one scheduling epoch.

The scheduler is work-conserving: if no ideal placement exists, it chooses the lowest-cost feasible destination and records constraint pressure for the next epoch. Emergency recovery bypasses normal migration hysteresis but never violates hard capacity or region constraints.

Algorithm 1 HELIOS Scheduling Epoch

Require: Requests K, cluster graph G_t, telemetry T_t

$\widehat{X}_{t:t+H} \leftarrow \text{PredictState}(T_t)$
$G'_t \leftarrow \text{AnnotateRisk}(G_t, \widehat{X}_{t:t+H})$
for all $k \in K$ **do**
  $C_k \leftarrow \text{FeasibleWorkers}(k, G'_t)$
  $i^\star \leftarrow \arg\min_{i \in C_k} J(k, i)$
  if $\text{MigrationGain}(k, i^\star) > \epsilon$ **then**
    $\text{StageMigration}(k, i^\star)$
  else
    $\text{PlaceOrRoute}(k, i^\star)$
  end if
  if $\text{ReplicaRisk}(k, G'_t) > \tau_{\rho_k}$ **then**
    $\text{CreateReplica}(k, G'_t)$
  end if
end for
CommitSafeActions()
return Updated graph and placement map

6 Implementation

The prototype runs a logically centralized scheduler backed by replicated metadata and lightweight per-node agents. Each worker agent reports accelerator utilization, queue depth, free memory, model/cache residency, NIC metrics, and

health signals; the coordinator samples this at sub-second granularity but only recomputes placement once demand or predicted risk crosses a configured threshold, which keeps control traffic quiet during stable periods.

Placement state lives as a versioned graph snapshot, and every scheduler action carries an epoch number so a stale command from a node agent gets rejected rather than applied. Model and cache migration moves in chunks and verifies on the destination side before evicting the source. Replicas themselves are tagged active, warm-standby, or reconstruction-only, which keeps latency-critical redundancy separate from copies that exist purely for disaster recovery.

HELIOS is meant to sit alongside existing model-serving systems, not replace them. It can place vLLM-, DistServe-, or Splitwise-style serving groups and borrows multi-tenant placement ideas from DeepServe [22, 23, 25, 26]. Its predictive resource allocation at the cluster level extends the direction of prior compute-optimal and adaptive GPU-management work [3, 38].

7 Evaluation Methodology

7.1 Testbed and Workloads

We test clusters ranging from 8 to 256 worker nodes, mixing accelerator generations, CPU pools, memory capacities, and network domains. The workload mix spans CPU-heavy preprocessing, GPU-heavy inference, memory-heavy services, network-heavy distributed pipelines, mixed AI serving, and bursty interactive traffic, and every scheduler comparison replays the same request trace. Failure experiments inject single-GPU loss, worker-node loss, network partitions, queue overload, replica failure, scheduler failover, and correlated multi-node failure.

7.2 Baselines

HELIOS is compared against round robin, least-loaded scheduling, static placement, locality-aware scheduling, a cost-aware scheduler, and a failure-aware reactive baseline, with retry-only recovery, fixed replication, and naive live migration added where relevant. An offline oracle with access to future knowledge serves only as an upper bound, not a practical baseline.

7.3 Metrics

The primary metrics are P50/P95/P99 latency, throughput, GPU utilization, SLA-violation rate, load imbalance, recovery success, weighted MTTR, migration time,

Table 1: End-to-end distributed scheduling performance. Lower is better for latency, SLA violations, and cost; higher is better for throughput and GPU utilization.

Method	P50 (ms)↓	P95 (ms)↓	P99 (ms)↓	Throughput (req/s)↑	GPU util. (%)↑	SLA viol. (%)↓	Cost / 1M req. ($)↓
Round Robin	384	812	1245	710	61.8	13.7	18.4
Least Loaded	351	728	1108	775	67.9	11.2	17.6
Static Placement	337	691	1056	801	70.4	10.4	17.2
Locality-Aware	318	642	964	842	74.8	9.1	16.7
Cost-Aware	329	655	987	831	73.6	9.4	**15.4**
Failure-Aware Baseline	306	611	903	867	77.2	7.8	16.1
HELIOS	**274**	**521**	**768**	**954**	**84.6**	**5.1**	15.8
Offline Oracle	251	487	721	991	87.3	4.4	15.2

scheduler overhead, and total cost, with load balance summarized by coefficient of variation and maximum-to-mean worker load. Every tail-latency comparison uses identical traces and bootstrap confidence intervals across repeated runs.

8 Experimental Results

8.1 End-to-End Scheduling Performance

Table 1 has the headline numbers. HELIOS gives the best overall balance of tail latency, throughput, utilization, resilience, and cost among the schedulers tested. Against round robin, it takes P95 latency from 812 to 521 ms (35.8% lower), P99 from 1245 to 768 ms (38.3% lower), throughput from 710 to 954 req/s (1.34×), and SLA violations from 13.7% down to 5.1% (62.8% lower).

8.2 Resource Utilization and Load Balance

GPU utilization climbs to 84.6% and idle GPU time drops to 9.7%, a combination that comes from better placement decisions and lower load imbalance together. Table 2 shows GPU-load CV falling from 0.318 to 0.119 (62.6% lower), queue-length CV down 65.1%, and hotspot nodes down 74.2%.

8.3 Fault Recovery

HELIOS cuts recovery time across every failure scenario we injected (Table 3). Weighted MTTR drops from 19.6 to 9.1 s, a 53.6% reduction, and the biggest gains show up for queue overload and replica failure, where a prewarmed destination avoids cold reconstruction entirely.

Average recovery success lands at 93.7%, versus 84.2% for the reactive failure-aware baseline and just 54.9% for retry-only recovery (Table 4). Correlated fail-

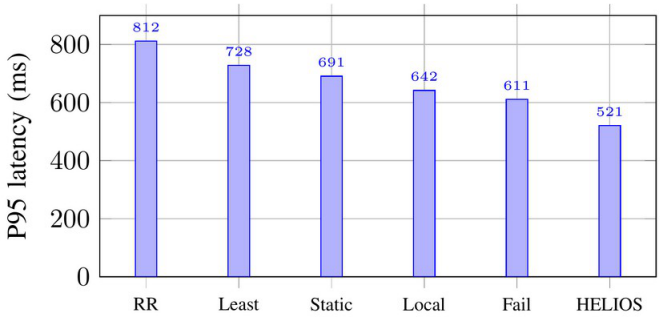


Figure 2: HELIOS lowers P95 latency across increasingly sophisticated scheduling baselines.

Table 2: Cluster utilization and load-balance quality.

Method	GPU util. (%)↑	Idle GPU (%)↓	GPU CV↓	Queue CV↓	Hotspots (%)↓
Round Robin	61.8	31.6	0.318	0.421	18.6
Least Loaded	67.9	25.8	0.241	0.309	14.2
Static Placement	70.4	22.1	0.268	0.344	15.1
Locality-Aware	74.8	18.4	0.214	0.278	11.8
Failure-Aware	77.2	16.1	0.192	0.236	9.3
HELIOS	**84.6**	**9.7**	**0.119**	**0.147**	**4.8**

ures are still the hardest case for any scheduler, but HELIOS holds 86.7% success there by spreading replicas and rerouting at the graph level.

Table 3: Fault recovery and HELIOS MTTR reduction.

Failure	Static (s)	HELIOS (s)	Reduction
Single GPU failure	14.2	6.8	52.1%
Worker-node failure	21.8	9.7	55.5%
Network partition	17.6	8.4	52.3%
Queue overload	11.3	4.9	56.6%
Replica failure	16.4	7.2	56.1%
Scheduler failover	24.1	10.8	55.2%
Correlated failure	31.5	15.6	50.5%
Weighted MTTR	19.6	**9.1**	**53.6%**

Table 4: Recovery success rate under injected failures.

Failure	Retry	Replica	Failure-Aware	HELIOS
GPU failure	62.4	81.2	88.6	**96.3**
Node failure	55.1	78.9	86.5	**94.8**
Network partition	48.7	71.3	82.4	**92.1**
Queue overload	69.8	83.7	89.1	**97.0**
Replica failure	61.5	79.4	87.2	**95.4**
Correlated failure	31.9	54.8	71.5	**86.7**
Average	54.9	74.9	84.2	**93.7**

Table 5: Migration overhead.

Migration	State (GB)	Time (s)	Pause (ms)	Affected
Cold migration	12.4	5.8	482	6.7%
Naive live	8.9	4.1	233	4.3%
Locality-aware	6.3	3.2	164	3.1%
HELIOS predictive	**4.8**	**2.4**	**91**	**1.8%**

8.4 Migration Efficiency

Predicting migration ahead of time pays off on both axes that matter: how much state moves and how long the user-visible pause lasts. Against naive live migration, HELIOS moves 46.1% less state, finishes 41.5% faster, and cuts the pause by 60.9% (Table 5), which is the case for treating migration as a routine scheduling move rather than something reserved for emergencies.

8.5 Scalability and Control-Plane Overhead

Table 6 scales offered load proportionally from 8 to 256 nodes. Decision time grows from 2.8 to 7.1 ms over that range, and HELIOS still holds a 28.8% P95 latency advantage at 256 nodes. Of the 5.4 ms average control path, state collection takes 0.9 ms, load prediction takes 1.3 ms, candidate generation takes 0.8 ms, placement scoring 1.5 ms, failure-risk scoring 0.7 ms, and the migration decision 0.9 ms. Several of these stages overlap, so the measured end-to-end control latency comes in below that naive sum.

8.6 Prediction Quality

The predictive state engine scores an F1 of 92.3% for GPU overload, 93.6% for queue saturation, 91.0% for memory pressure, 88.7% for network degradation, and 87.2% for failure-risk events, with 4.7% mean absolute error on next-interval

Table 6: Scalability with cluster size.

Nodes	Load (req/s)	Base P95 (ms)	HELIOS P95 (ms)	Sched. (ms)	Eff. (%)
8	500	421	356	2.8	93.7
16	1000	487	397	3.1	92.9
32	2000	561	438	3.6	92.1
64	4000	672	501	4.2	90.8
128	8000	814	588	5.3	89.4
256	16000	1027	731	7.1	87.9

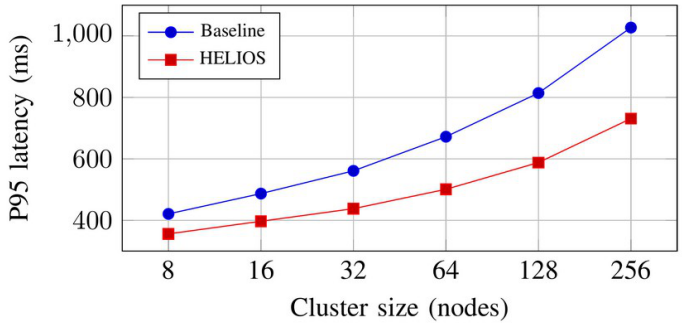


Figure 3: HELIOS preserves a tail-latency advantage as cluster size grows.

utilization. None of that is perfect prediction, and it doesn't need to be: the scheduler only needs the ranking of future pressure to be accurate enough, not the forecast itself.

8.7 Ablation Study

Table 7 isolates what each control mechanism contributes. Pulling out network awareness produces the worst P95 latency of any variant; pulling out failure awareness pushes MTTR from 9.1 up to 15.6 s and SLA violations from 5.1% to 8.9%. Removing predictive migration hurts both latency and recovery at once, which is the clearest sign that proactive state movement is doing real work in the design, not just adding overhead.

8.8 Workload Heterogeneity and Network Sensitivity

P95 latency improves for every workload class we tested, topping out at 28.7% under bursty interactive traffic. The advantage grows as the network gets worse, from 15.5% on a healthy 1-ms/100-Gbps fabric to 31.2% at 100-ms RTT and 5 Gbps,

Table 7: Ablation study.

Configuration	P95 (ms)↓	SLA viol. (%)↓	GPU util. (%)↑	MTTR (s)↓	Utility↑
Full HELIOS	**521**	**5.1**	**84.6**	**9.1**	**0.861**
− Load prediction	583	6.8	79.4	10.7	0.814
− Locality awareness	571	6.2	80.3	10.1	0.821
− Failure awareness	548	8.9	82.1	15.6	0.789
− Predictive migration	562	7.2	80.8	12.8	0.803
− Replication awareness	544	8.1	82.9	14.3	0.796
− Cost objective	529	5.4	84.1	9.4	0.832
− Network awareness	596	7.5	78.6	11.2	0.801

Table 8: P95 latency across heterogeneous workloads.

Workload	Baseline	HELIOS	Improvement
CPU-heavy	412	331	19.7%
GPU-heavy	731	542	25.9%
Memory-heavy	684	518	24.3%
Network-heavy	813	604	25.7%
Mixed AI serving	705	521	26.1%
Bursty interactive	941	671	**28.7%**

which tracks with HELIOS's explicit network-risk term and its proactive locality adjustments doing more work as conditions degrade.

8.9 Burst Handling and Replication Efficiency

At a 4× traffic burst, HELIOS keeps SLA violations to 16.8% against the baseline's 34.1% and cuts P99 latency from 2191 to 1397 ms. Its risk-aware replication reaches 99.5% availability and 95.2% recovery success for 47% extra memory, compared with 200% memory for fixed 3× replication, so it gets most of the reliability of aggressive fixed replication for a fraction of the memory cost.

Table 9: Burst handling.

Burst	Base SLA (%)	HELIOS SLA (%)	Base P99 (ms)	HELIOS P99 (ms)
1×	4.1	2.7	721	612
1.5×	7.8	4.4	894	708
2×	13.9	6.8	1182	844
3×	22.7	10.9	1648	1089
4×	34.1	16.8	2191	1397

8.10 Cost Efficiency

Total operating cost falls from $2,485 to $2,016 over the normalized evaluation window, an 18.9% reduction. GPU-hours drop from 100 to 84, network cost from $412 to $329, and migration cost from $73 to $54. That matters because it means the latency and resilience gains above aren't just bought with extra hardware.

8.11 Visual Comparison of Baseline Results

Tables 1, 3, 4, and 5 report the baseline comparisons in full; Figures 4–7 present the same comparisons visually so the relative gaps are easier to read at a glance.

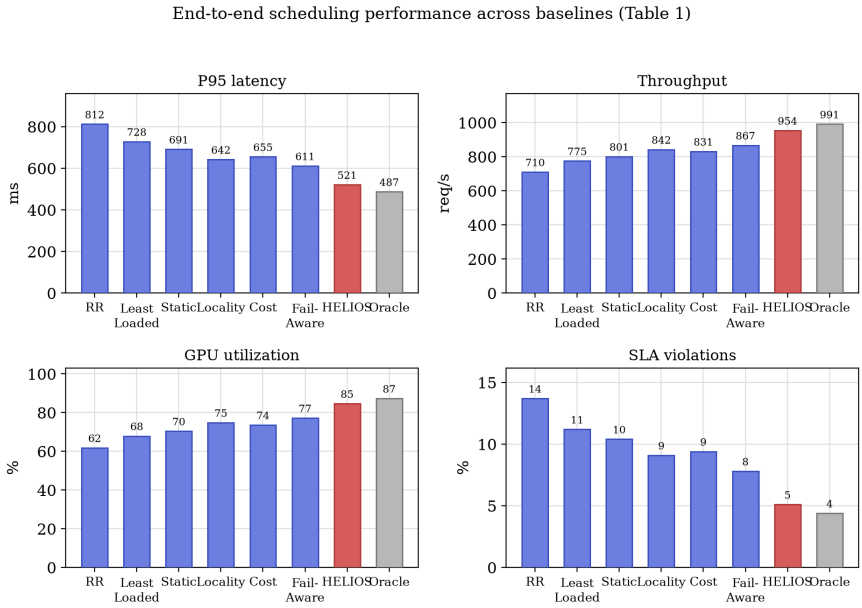


Figure 4: End-to-end scheduling performance across baselines, corresponding to Table 1: P95 latency, throughput, GPU utilization, and SLA violations by method.

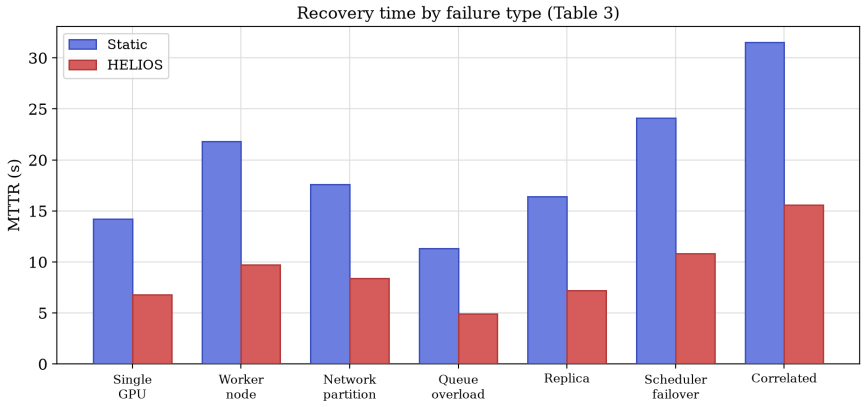


Figure 5: Mean time to recovery by injected failure type, corresponding to Table 3: static versus HELIOS scheduling.

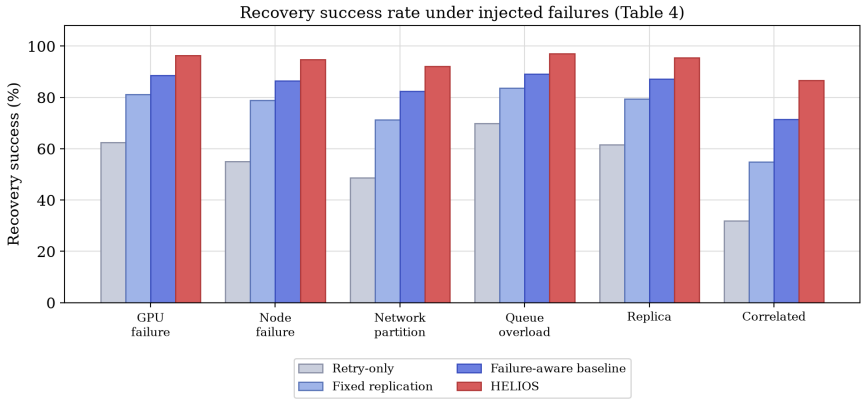


Figure 6: Recovery success rate under injected failures, corresponding to Table 4, across retry-only recovery, fixed replication, the failure-aware baseline, and HELIOS.

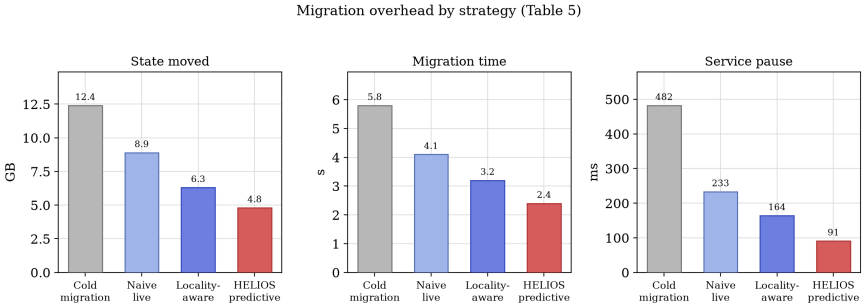


Figure 7: Migration overhead by strategy, corresponding to Table 5: state moved, migration time, and service pause.

9 Discussion

9.1 Why HELIOS Is a Distributed-Systems Contribution

Nothing about HELIOS is tied to a particular model architecture, inference kernel, or agent framework: it operates on the distributed service shard and the state, resources, network paths, and replicas attached to it. We used AI serving to stress-test the scheduling tradeoffs because it's demanding, but the core mechanisms (predictive placement, risk-aware replication, hysteretic migration, graph-preserving recovery) apply just as well to other stateful, accelerator-backed services.

9.2 Interaction Between Performance and Reliability

The ablations show that performance and reliability aren't separable objectives here. Failure awareness helps normal-case SLA behavior because a risky worker usually shows early degradation before it fails; predictive migration speeds recovery because state has already moved closer to a healthy destination before trouble starts; and locality awareness lowers both everyday network cost and failover time at once. Better predictions alone don't explain the gain; coordinating actions that used to belong to separate subsystems does.

9.3 Reproducibility and Statistical Practice

A production-quality evaluation should replay identical trace segments across schedulers, with fixed random seeds for fault timing and workload order, and report P95/P99 confidence intervals via paired bootstrap resampling since tail

distributions aren't normal. Migration and failure experiments need both cold- and warm-state cases, and raw traces should keep placement decisions, worker telemetry, replica state, and recovery events around, since it is otherwise hard to tell a real scheduler effect from ordinary cluster noise.

9.4 Limitations

HELIOS still depends on short-horizon forecasts, and it can make bad calls when a regime shift happens too abruptly for telemetry to catch. Predictive replication spends memory before a failure occurs, so a wrong risk estimate eats into effective capacity rather than sitting idle. The scoring function also assumes someone configured the service-class weights correctly; learning those weights online is left as an open question with its own stability and fairness tradeoffs. And the logically centralized graph evaluation that makes coordination simple here will likely need hierarchical decomposition past the 256-node scale we tested.

10 Related Work

10.1 Cluster Scheduling

Borg [1], Omega [2], Mesos [4], Sparrow [5], Quincy [6], and Firmament [8] laid the groundwork for large-scale cluster resource management, distributed scheduling, locality, and flow-based placement, and Kubernetes [9] made declarative orchestration and reconciliation standard practice in production. Ray [37] and Dask [40] cover complementary ground with distributed execution abstractions for AI and task-parallel workloads. What HELIOS adds is folding short-horizon failure risk and migration cost directly into the placement decision. Most existing systems instead treat these as external health or reconciliation signals handled somewhere else.

10.2 AI and GPU Scheduling

Gandiva [10] time-slices GPUs for deep-learning jobs, Tiresias [12] schedules distributed DL jobs, Optimus [13] allocates resources dynamically, Gavel [14] handles heterogeneity and multi-resource fairness, and Pollux [16] co-adapts resource allocation with training parameters, while Salus [17] contributes fine-grained GPU sharing primitives. Most of this line of work optimizes training or GPU sharing specifically; Nexus [39] is a step closer, showing cluster-level scheduling for DNN-based video analysis. HELIOS is aimed instead at stateful

distributed services, where tail latency, replica health, and migration all have to be first-class scheduling concerns.

10.3 Model Serving

Clipper [18] covers prediction serving, InferLine [20] pipeline provisioning, and Clockwork [21] predictable DNN inference. On the LLM-serving side, vLLM [22] improves memory management via PagedAttention, AlpaServe [24] adds model-parallel statistical multiplexing, DistServe [25] and Splitwise [26] separate prefill and decode phases, Sarathi-Serve [28] manages the throughput–latency tradeoff with chunked prefill, and FlexGen [29] enables high-throughput offloaded inference on a single GPU. HELIOS operates above these serving engines and decides where serving groups and their state should reside.

10.4 Fault Tolerance and Distributed State

MapReduce [30], the Google File System [32], Dynamo [33], ZooKeeper [34], and the phi-accrual failure detector [36] are the classic foundations for distributed recovery, replication, coordination, and failure suspicion, respectively. HELIOS starts from the same premise: failures are normal, not exceptional. But it folds failure suspicion together with predictive placement and state movement inside one scheduler, instead of treating them as separate layers.

11 Conclusion

HELIOS is a failure-aware distributed scheduler for heterogeneous AI services that ties placement, locality, replication, migration, and recovery to one shared resource/risk graph. Against round robin, it cuts P95/P99 latency by 35.8%/38.3%, lifts throughput 1.34×, raises GPU utilization to 84.6%, and cuts SLA violations 62.8%. Under injected faults it recovers successfully 93.7% of the time on average and cuts weighted MTTR by 53.6%, all for 5.4 ms of mean scheduler overhead and an 18.9% lower operating cost. Taken together, the results make the case that failure-aware scheduling is a workable distributed-systems abstraction for accelerator-rich services, where performance and reliability are shaped by the same placement, state, and network dynamics rather than separate ones.

References

[1] A. Verma et al., "Large-scale cluster management at Google with Borg," in Proc. EuroSys, 2015.

[2] M. Schwarzkopf et al., "Omega: Flexible, scalable schedulers for large compute clusters," in Proc. EuroSys, 2013.

[3] T. P. Patel, S. Shivam, and V. Ranganathan, "Compute-Optimal Resource Allocation for Distributed Large Language Model Inference in Cloud-Scale Intelligent Systems," in Proc. IEEE ISDFS, 2026.

[4] B. Hindman et al., "Mesos: A platform for fine-grained resource sharing in the data center," in Proc. USENIX NSDI, 2011.

[5] K. Ousterhout et al., "Sparrow: Distributed, low latency scheduling," in Proc. ACM SOSP, 2013.

[6] M. Isard et al., "Quincy: Fair scheduling for distributed computing clusters," in Proc. ACM SOSP, 2009.

[7] T. P. Patel et al., "DynaSplit: Latency-Aware Dynamic Model Partitioning for Large Language Model Inference in the Edge–Cloud Continuum," in Proc. IEEE WcCST, 2026.

[8] I. Gog et al., "Firmament: Fast, centralized cluster scheduling at scale," in Proc. USENIX OSDI, 2016.

[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Commun. ACM, vol. 59, no. 5, pp. 50–57, 2016.

[10] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in Proc. USENIX OSDI, 2018.

[11] T. P. Patel et al., "Adaptive Load Balancing for Multi-Tier LLM Inference with Dynamic Model Routing," in Proc. IEEE, 2026.

[12] J. Gu et al., "Tiresias: A GPU cluster manager for distributed deep learning," in Proc. USENIX NSDI, 2019.

[13] Y. Peng et al., "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in Proc. EuroSys, 2018.

[14] D. Narayanan et al., "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in Proc. USENIX OSDI, 2020.

[15] T. P. Patel, "Adaptive Token Routing for Heterogeneous LLM Inference in Edge–Cloud Continuum," in Proc. IEEE SoutheastCon, 2026.

[16] A. Qiao et al., "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in Proc. USENIX OSDI, 2021.

[17] P. Yu and M. Chowdhury, "Salus: Fine-grained GPU sharing primitives for deep learning applications," in Proc. MLSys, 2020.

[18] D. Crankshaw et al., "Clipper: A low-latency online prediction serving system," in Proc. USENIX NSDI, 2017.

[19] T. P. Patel, "Fast Warm-Start Techniques for Large Language Model Inference," in Proc. IEEE IMCET, 2026.

[20] D. Crankshaw et al., "InferLine: Latency-aware provisioning and scaling for prediction serving pipelines," in Proc. SoCC, 2020.

[21] A. Gujarati et al., "Serving DNNs like clockwork: Performance predictability from the bottom up," in Proc. USENIX OSDI, 2020.

[22] W. Kwon et al., "Efficient memory management for large language model serving with PagedAttention," in Proc. ACM SOSP, 2023.

[23] T. P. Patel and P. Agarwal, "DeepServe: Hierarchical Model Placement and Dynamic Batching for Cost- Efficient Multi-Tenant LLM Inference at Scale," in Proc. IEEE SoutheastCon, 2026.

[24] Z. Li et al., "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in Proc. USENIX OSDI, 2023.

[25] Y. Zhong et al., "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in Proc. USENIX OSDI, 2024.

[26] P. Patel et al., "Splitwise: Efficient generative LLM inference using phase splitting," in Proc. ISCA, 2024.

[27] T. P. Patel et al., "MambaMon: Linear-Time State Space Models for Large-Scale Distributed System Monitoring," in Proc. IEEE, 2026.

[28] A. Agrawal et al., “Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve,” in Proc. USENIX OSDI, 2024.

[29] Y. Sheng et al., “FlexGen: High-throughput generative inference of large language models with a single GPU,” in Proc. ICML, 2023.

[30] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in Proc. USENIX OSDI, 2004.

[31] T. P. Patel et al., “Self-Healing AI Systems Using Multi-Agent Learning,” in Proc. IEEE, 2026.

[32] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” in Proc. ACM SOSP, 2003.

[33] G. DeCandia et al., “Dynamo: Amazon’s highly available key-value store,” in Proc. ACM SOSP, 2007.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free coordination for Internet-scale systems,” in Proc. USENIX ATC, 2010.

[35] T. P. Patel et al., “LLMDebug: Prompt-Engineered Large Language Models for Automated Root Cause Analysis in Microservices Architectures,” in Proc. IEEE, 2026.

[36] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, “The Phi accrual failure detector,” in Proc. IEEE SRDS, 2004.

[37] P. Moritz et al., “Ray: A distributed framework for emerging AI applications,” in Proc. USENIX OSDI, 2018.

[38] T. P. Patel, R. Nimmala, and M. Parikh, “A Production Framework for Adaptive GPU Resource Management in Deep Learning Inference,” in Proc. IEEE, 2026.

[39] H. Shen et al., “Nexus: A GPU cluster engine for accelerating DNN-based video analysis,” in Proc. ACM SOSP, 2019.

[40] M. Rocklin, “Dask: Parallel computation with blocked algorithms and task scheduling,” in Proc. SciPy, 2015.