ElastiCo: Elastic Configuration and Interference-Aware Orchestration for GPU Clusters

Jinghao Wang[a], Yihang Zhou[b], Xiaoyang Sun[c,*], Chunming Hu[a], Tianyu Wo[a], Xu Wang[a], Albert Y. Zomaya[d] and Renyu Yang[a]

[a]School of Software, Beihang University, Beijing, 100083, China
[b]School of Computer Science and Engineering, Beihang University, Beijing, 100083, China
[c]School of Computer Science, University of Leeds, Leeds, LS2 9JT, United Kingdom
[d]School of Computer Science, University of Sydney, Sydney, Australia

ARTICLE INFO

Keywords:
GPU Cluster Management
Job Co-Location
Interference-Aware Scheduling

ABSTRACT

Modern GPU clusters must simultaneously serve deep learning training and offline large language model inference workloads, yet existing schedulers treat these as isolated resource consumers with rigid, static allocations. This leaves substantial GPU capacity underutilized: training jobs reserve entire devices despite periodic idle phases, while offline inference tasks over-provision GPUs despite bursty demand patterns. We present ElastiCo, an elastic co-location framework that enables training and inference workloads to safely share GPUs through three integrated mechanisms. First, *Resource Shape Transformation* exposes each job as a family of feasible resource-performance configurations. Second, *Elastic Shadow Pricing* decomposes the resulting multi-resource allocation problem into per-job configuration selection subproblems via dynamic per-resource shadow prices. Third, *Interference-Aware Co-location* uses a predictor trained on hardware-counter and task-level features to estimate pairwise performance degradation under GPU sharing. Implemented as native Kubernetes middleware requiring no user-code modifications, ElastiCo is evaluated on a 64-GPU testbed and through large-scale trace-driven simulations (up to 512 GPUs), reducing the average JCT by up to 2.94×, increasing the cluster throughput by 2.02×, and increasing the GPU utilization from approximately 25% to 46%.

1. Introduction

GPU clusters are the single largest upfront investment in contemporary AI infrastructure, yet real-world production data repeatedly expose a challenge: even under nominal over-subscription, average GPU utilization almost never exceeds 30% [1, 2, 3]. This kind of waste is not an issue of workload, but a system *architectural* design. Modern compute clusters are increasingly required to handle two primary types of workloads: deep learning (DL) training, which uses large numbers of GPUs for many hours or even days to run iterative gradient descent, and *offline LLM inference*, which covers batch-style jobs such as large-scale data labeling, model evaluation, synthetic data creation, etc. [4, 2]. In contrast to latency-sensitive online serving, offline inference focuses on maximizing overall throughput rather than minimizing the latency of individual requests, but it still requires substantial GPU compute and memory. As demand for both types of workload increases, the cluster must handle them at the same time, turning efficient resource sharing into a primary infrastructure challenge.

The primary reason for this underutilization is *static resource partitioning*: operators allocate distinct GPU pools for training and inference, and provision each pool according to its own peak demand. Existing studies [1, 3] show that, in real production clusters, the average utilization of GPU SMs typically stays well below 30%. An increasing number of studies show that running training and inference together on the same GPUs can greatly boost utilization, because these two types of workloads have bursty, mostly non-overlapping resource demands that introduce exploitable idle periods [5, 6, 7, 8]. However, fully unlocking this potential demands overcoming three tightly interrelated challenges that current systems address only in isolation.

*Corresponding author
wang_jinghao@buaa.edu.cn (J. Wang); zhou_yihang@buaa.edu.cn (Y. Zhou); X.Sun4@leeds.ac.uk (X. Sun); hucm@buaa.edu.cn (C. Hu); woty@buaa.edu.cn (T. Wo); xuwang@buaa.edu.cn (X. Wang); albert.zomaya@sydney.edu.au (A.Y. Zomaya); renyuyang@buaa.edu.cn (R. Yang)
ORCID(s):

Challenge-1: Existing schedulers assume that each job has a fixed resource requirement and thus fail to leverage the significant *configuration elasticity* that DL workloads naturally provide. For instance, a training job can exchange increased computation for reduced memory usage by using activation checkpointing. *Challenge-2: Existing cluster schedulers*, like Pollux [9] and Lucid [10], *focus solely on optimizing GPU allocation or packing for training jobs*, and do not incorporate inference workloads or co-location interference into their models. *Challenge-3: Existing interference predictors for pairwise co-location [7, 1, 5, 6] are designed for specific co-location patterns and fail to capture cluster-wide, multi-resource optimization*. Heuristic approaches also exhibit poor scalability as the configuration space expands. Moreover, even with carefully selected configurations and allocations, co-located workloads still compete for shared hardware resources, such as streaming multiprocessors, memory bandwidth, and on-device memory.

These three challenges are interdependent: static configuration constrains which co-location setups are even possible, allocation complexity affects whether the best setup is selected, and interference awareness guarantees that the chosen setups remain safe during execution. Addressing any challenge on its own delivers only limited benefits because the remaining unresolved challenges quickly become the dominant bottlenecks.

We present ElastiCo, an elastic co-location framework that simultaneously tackles all three of these challenges. The rationale is that configuration flexibility, resource allocation, and interference awareness are interdependent aspects of one optimization problem and therefore need to be designed jointly. ElastiCo introduces *Resource Shape Transformation* (RST, §3.1) to expose each job as a family of feasible resource–performance profiles, *Elastic Shadow Pricing* (ESP, §3.2) to solve the resulting multi-resource allocation via cost/price-guided constraints, and *Interference-Aware Co-location* (IAC, §3.3) to predict and bound pairwise degradation. *Phase-Aware Disaggregated Scheduling* (PDS, §3.4) orchestrates these components in a closed-loop control cycle, coordinating queue-aware capacity reservation, configuration switching, and preemptive migration.

ElastiCo is realized as Kubernetes-native middleware and operates without requiring any changes to user code. ElastiCo is evaluated on a 64-GPU A100 testbed and via large-scale trace-driven simulations (scaling up to 512 GPUs). The results indicate that ElastiCo reduces average job completion time (JCT) by up to 2.94×, boosts cluster throughput by 2.02×, increases GPU utilization from about 25% to 46%, and decreases the number of extra GPU instances needed for concurrent offline inference by 44%.

To conclude, this paper offers the following key contributions:

1. **Resource Shape Transformation (RST)**, a method that models each job as a family of resource-performance profiles by exploring systematic settings, such as activation checkpointing, micro-batch sizing, mixed-precision modes, and KV-cache configurations (§3.1). To our knowledge, RST is the first approach that treats intra-job configuration flexibility as a first-class scheduling dimension for job co-location.
2. **Elastic Shadow Pricing (ESP)**, a cost-based mechanism that decomposes joint configuration selection and multi-resource assignment into per-job subproblems with interference penalties, scaling to hundreds of jobs (§3.2).
3. **Interference-Aware Co-location (IAC)**, an interference predictor over hardware-counter and task-level features, combined with adaptive tolerance thresholds, that enables interference-aware co-location admission and preemptive migration to preserve per-job performance under aggressive GPU sharing (§3.3).
4. **Production-Grade Implementation**, a Kubernetes-native middleware with non-intrusive profiling via a two-dimensional optimized profiler, framework-native memory budget coordination (§4), and a comprehensive evaluation demonstrating up to 2.94× JCT reduction, 2.02× throughput improvement, and 44% fewer GPU instances compared to static partitioning (§5).

2. Background and Motivation

This section measures the resource inefficiencies of current GPU cluster management, highlights the resulting opportunities for workload co-location, and outlines the design challenges that ElastiCo is built to overcome.

2.1. GPU Underutilization in Production

Two real-world production datasets have been made available: the Alibaba PAI cluster [2] and a stable diffusion serving platform [11]. They provide two months of training and inference traces, in terms of GPU SM utilization.

Figure 1 shows the cumulative distribution of GPU SM utilization. In the PAI cluster, training jobs (which request a fixed number of GPUs) reach an average utilization of only 14.3%, and nearly 90% of them operate below 50%. Inference jobs (which obtain short-term GPUs on demand) perform even worse, with a mean SM utilization of just 6.4%. Overall, the cluster-wide average is 10.5%, indicating that most allocated GPU cycles remain unused. The GenAI

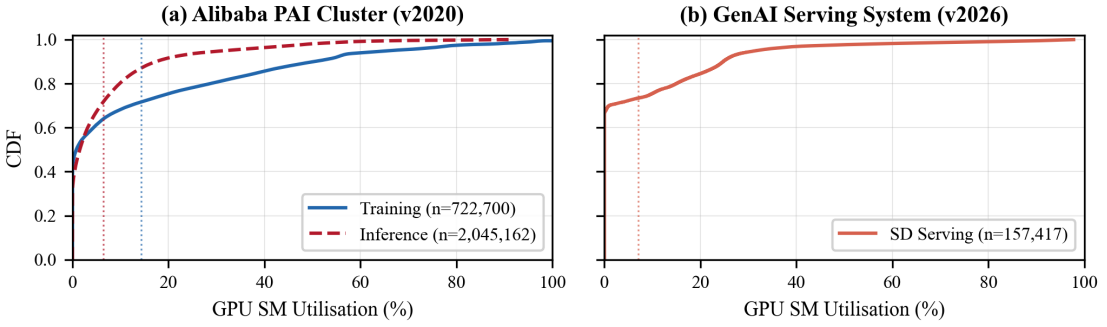


Figure 1: CDF of GPU SM utilization across two production systems. (a) Alibaba PAI Cluster [2]: training vs. inference jobs. (b) GenAI Serving System [11]: per-pod time-series samples.

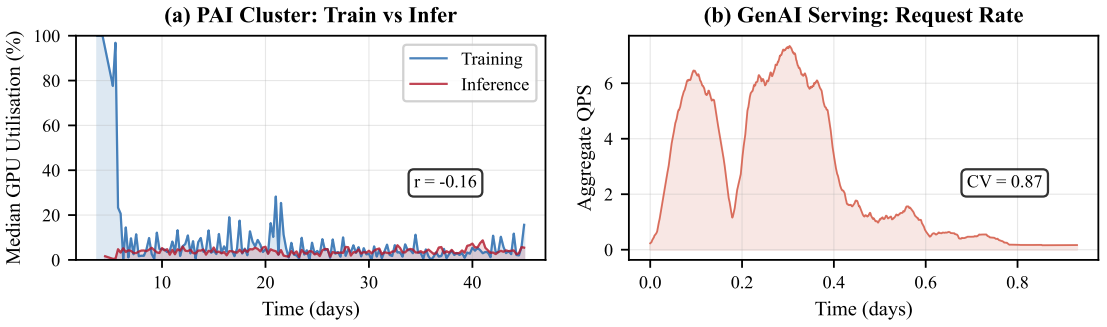


Figure 2: Temporal demand characteristics. (a) Median GPU utilization in the PAI cluster, aggregated over 6-hour intervals; (b) Total QPS of the GenAI serving infrastructure.

serving system follows a comparable trend: its GPU pods achieve an average utilization of 7.0%, with 97.7% of time-series measurements under 50% and a median that is effectively zero, pointing to long idle periods between request bursts. These results align with recent industry-wide reports that production GPU clusters typically exhibit average utilizations below 30% [1, 3].

This waste stems from several sources, such as naturally bursty workloads, peak-based resource provisioning, and - most importantly - *static pool partitioning*, which partitions training and inference into distinct GPU pools. Because low usage in one pool cannot offset surges in the other, a large portion of reserved hardware remains unused, even as other jobs are left without enough compute.

2.2. Divergent Demand Patterns

A co-location strategy can exploit this underutilization only when the idle periods are *accessible*. As demonstrated, training and inference workloads have fundamentally different demand profiles: both are bursty, but in distinct ways, creating exactly this kind of exploitable slack.

Figure 2(a) shows the median GPU utilization of training and inference jobs in the PAI cluster over 6-hour intervals across a representative 45-day period. Training experiences an early spike (around 90%) followed by a long tail of low utilization (5–15%), while inference stays consistently below 10%. As a result, neither class maintains high utilization for long, and GPUs remain underused for most of the trace. The Pearson correlation between the two time series is weakly negative ($r = -0.16$, $p < 0.05$), indicating only a mild inverse association. Importantly, co-location does not rely on strong anti-correlation; it is enough that the two workloads *infrequently peak together*, so that one pool typically has slack when the other is heavily loaded.

The GenAI serving trace supports this conclusion. Figure 2(b) shows the aggregate request rate (QPS) for the Stable Diffusion system, which varies sharply, with a coefficient of variation of 0.87. The peak-to-trough ratio is nearly 85× (comparing the 95th to the 5th percentile of non-zero QPS), and QPS frequently falls close to zero for prolonged

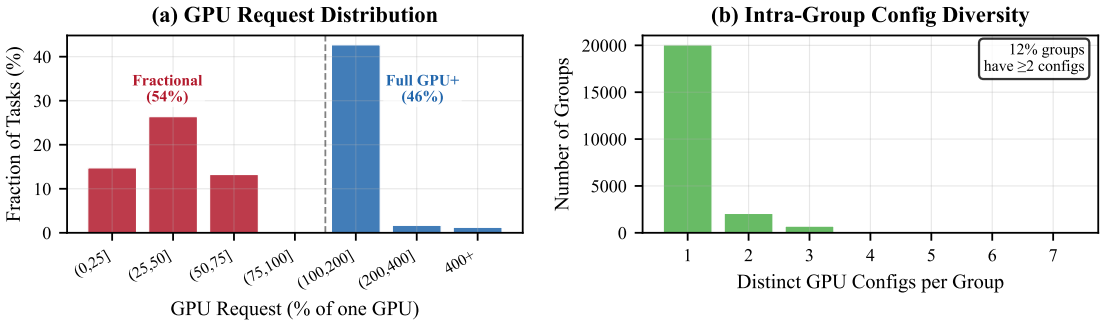


Figure 3: Job states in the PAI cluster. (a) Distribution of GPU requests; (b) Intra-group configuration diversity.

periods between bursts. Each such idle stretch is a period during which GPUs reserved for inference are completely unused - this is exactly the temporal slack.

2.3. Opportunity for Reconfigurability

Production traces show that workloads have substantial, but largely untapped, configuration flexibility. Bursty demand dictates *when* co-location is possible, while configuration elasticity determines *how much* co-location can occur.

Figure 3(a) presents the distribution of GPU requests in the PAI cluster. Over 54% of tasks request *fractional* GPUs, indicating that many workloads can run without exclusive access to a full GPU. Figure 3(b) examines workload *groups* - sets of jobs with identical scripts, parameters, and data sources - and shows that among recurring workload families (groups with ≥5 jobs), 12% use two or more distinct GPU configurations, with some spanning up to 7, demonstrating recurring configuration heterogeneity in production. While the group-level view highlights diversity across workload families, a complementary *user-level* analysis, Figure 4, estimates how many individual high-GPU tasks could instead run at smaller scales.

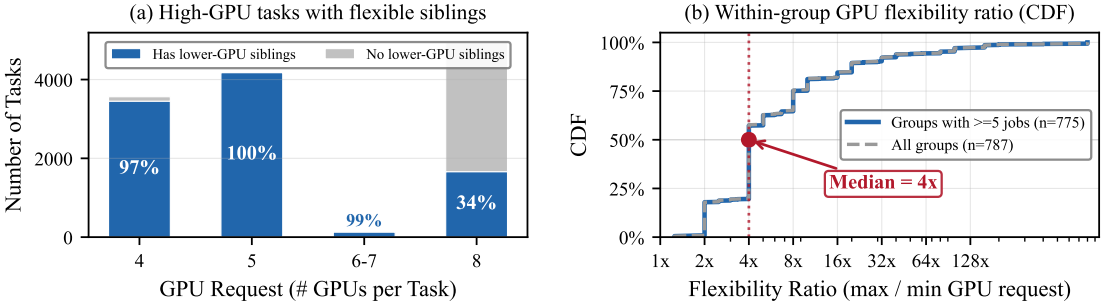


Figure 4: Job configurations in the PAI cluster. (a) GPU request vs usage. (b) CDF of within-group GPU flexibility ratio.

This diversity offers indirect evidence of configuration flexibility: when jobs in the same group succeed with varying GPU allocations, the workload likely supports multiple valid configurations. To characterize this, we focus on the 12,818 tasks that demand ≥4 GPUs and observe that 73% were submitted by users whose overall portfolios also contain jobs completed with fewer than 4 GPUs, shown in Figure 4(a). The median flexibility ratio per-user (max/min GPU request) is 4 times, shown in Figure 4(b), indicating that a typical recurring workload can run over roughly a 4× span of GPU counts. We further substantiate this through controlled profiling of 12 representative workloads in §5.1, showing that each supports 12-198 feasible configurations covering different batch sizes, parallelism schemes, and memory-optimization settings.

2.4. The Need for a Unified Orchestration Layer

Modern GPU clusters operate on two decoupled layers - application and infrastructure. At the application level, deep learning frameworks expose configuration knobs, such as batch sizes, checkpointing policies, and KV-cache management, that determine a workload's resource footprint and performance. At the infrastructure level, cluster schedulers allocate GPUs and other resources based on simplified, typically fixed resource requests. This decoupling introduces three systematic challenges that limit overall cluster utilization.

C.1 Configuration-Capacity Mismatch. Most production schedulers assume a job's resource request as static throughout its entire run. However, as the trace analysis above shows, many workloads can run correctly under several valid configurations that balance performance against hardware usage. When the scheduler overlooks this flexibility, jobs can sit in the queue even though sufficient resources are actually available. For instance, a distributed training job that requests eight GPUs for data parallelism may sit in the queue until all eight are available, even though the same job could run on four GPUs using tensor-pipeline parallelism with activation checkpointing, matching the 4× median flexibility ratio seen in the trace. Since conventional schedulers are unaware of such configuration choices, they miss these opportunities, causing longer wait times and poorer hardware utilization.

C.2 Combinatorial Allocation Complexity. Even when configuration choices are available, the allocation problem remains combinatorially difficult. The scheduler must simultaneously pick a configuration for every active job and distribute heterogeneous resources across potentially hundreds of concurrent workloads while maintaining both throughput and fairness. Existing elastic training schedulers [9, 10] optimize GPU allocation for training alone without modeling inference workloads or co-location constraints, while pairwise co-location approaches do not address cluster-wide multi-resource optimization.

C.3 Configuration-Blind Co-location. GPU sharing mechanisms like NVIDIA MPS enable multiple workloads to run simultaneously on one device, but current schedulers usually determine co-location feasibility using only static capacity metrics, most often GPU memory usage. In reality, however, interference between co-located workloads strongly depends on their execution configurations. Even if two jobs fit together in memory, they can still interfere heavily at runtime by contending for shared hardware. For example, activation recomputation can greatly increase GPU compute demand, and some tensor-parallel tasks put intense pressure on SMs and memory bandwidth. Co-locating such jobs without considering this interference can severely degrade performance or even cause failures. Thus, effective co-location must account for configuration-specific hardware usage patterns, not just memory capacity.

These three challenges reveal a core shortcoming of current cluster managers: their schedulers are unaware of the *configuration elasticity* of modern deep learning workloads. They cannot adapt job configurations to match available resources or anticipate interference among co-located workloads at runtime.

3. System Design

ElastiCo treats *configuration flexibility, resource allocation, and interference awareness* as interdependent aspects of a single optimization problem that must be designed together. Addressing any one alone turns the remaining two into hard constraints that limit utilization improvements. This leads to a four-module architecture (Figure 5), where each module handles one dimension and provides clear interfaces to the others:

- *Resource Shape Transformation (RST, §3.1)* exposes each job as a *family* of feasible resource–performance profiles, transforming rigid resource requests into a malleable configuration space.
- *Elastic Shadow Pricing (ESP, §3.2)* decomposes the combinatorial joint configuration-selection and multi-resource allocation problem into tractable per-job subproblems via a cost-efficient packing algorithm.
- *Interference-Aware Co-location (IAC, §3.3)* predicts pairwise performance degradation under GPU sharing and feeds interference costs back into ESP as penalties, internalizing co-location externalities.
- *Phase-Aware Disaggregated Scheduling (PDS, §3.4)* orchestrates the above modules in a closed-loop control cycle, coordinating runtime adaptation including queue-aware capacity reservation.

The remainder of this section formalizes each module. Table 1 summarizes the key notation used throughout.

3.1. Resource Shape Transformation (RST)

ElastiCo defines the formalization of each job's configuration space. For each job $j \in \mathcal{J}$, we define a configuration set $C_j = \{c_j^1, \ldots, c_j^k\}$. A configuration $c_j^i \in C_j$ is specified by a tuple of system-level knobs: $c_j^i = (s_1, s_2, \ldots, s_n)$, for example, the micro-batch size, activation checkpointing policy, and mixed-precision mode for a training job. The RST function $RST(c_j^i)$ translates each configuration into a resource–performance profile $p_j^i = (R_j(c), T_j(c), L_j)$, where

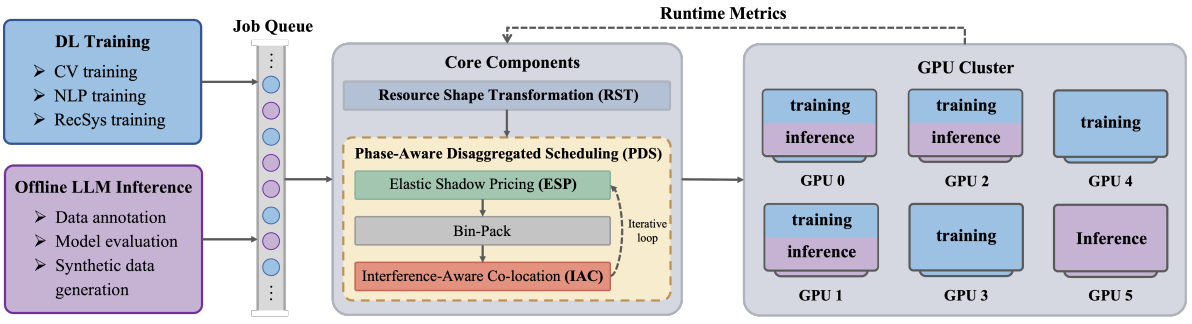


Figure 5: The architecture of ElastiCo.

Table 1
Notation used in the system design.

Symbol	Description
$\mathcal{J}$, $\mathcal{Q}$	Active job set, pending queue
$\mathcal{C}_j$	Configuration space of job j
$R_j(c)$, $T_j(c)$, L_j	Resource vector, throughput, minimum performance target
λ^t	Shadow-price vector at epoch t
$\hat{S}_{i\|j}$	Predicted slowdown of job i when co-located with j
$\tau_j(t)$	Adaptive tolerance threshold for job j
$\Gamma_j^{(m)}(c)$	Interference penalty at ESP–IAC round m
$\mathbf{C}$, $\mathbf{C}_{\text{eff}}$	Physical and effective cluster capacity

$R_j(c)$ is a multi-dimensional resource demand vector (GPU SMs, GPU memory, CPU cores, and host RAM), $T_j(c)$ is the estimated isolated performance (e.g., throughput per iteration or tokens generated per second), and L_j is the minimum performance target for job j (e.g., a throughput floor specified at submission time).

3.1.1. Configuration Space

ElastiCo divides incoming workloads into two types: *training jobs* and *offline inference jobs*. Because they differ in execution behavior and performance bottlenecks, their resource-performance tuning knobs are managed independently.

Training jobs: Training jobs iterate over three computational phases (i.e., forward and backward propagation, and parameter update) for thousands or millions of iterations until convergence. RST focuses on *system-level configuration knobs* that reshape the job's resource footprint without altering model semantics. The settings are as follows.

(i) Activation Recomputation [1]. Activation recomputation creates a tunable memory-compute trade-off: disabling recomputation maximizes throughput at the cost of higher GPU memory consumption, while enabling it significantly reduces the peak memory footprint by recomputing activations during the backward pass.

(ii) Micro-Batch Size. The micro-batch size is the number of training samples processed per GPU. It directly controls the trade-off between memory consumption and training throughput. Larger micro-batches increase arithmetic intensity and per-iteration throughput but demand proportionally more memory for activations and intermediate tensors; smaller micro-batches reduce peak memory at the cost of lower computational efficiency.

(iii) Automatic Mixed Precision (AMP). Mixed-precision training [13] executes the forward and backward passes in half-precision arithmetic (FP16 or BF16) while maintaining a master copy of the weights in FP32 for numerically stable gradient accumulation. Enabling AMP roughly halves the memory footprint of activations and intermediate tensors and increases arithmetic throughput on hardware equipped with dedicated half-precision units, with negligible impact on final model quality.

Offline inference jobs: Offline inference jobs, such as large-scale dataset annotation or model evaluation, prioritize aggregate throughput rather than per-request latency. Their performance is primarily determined by batching efficiency

[1] Activation recomputation (also known as activation checkpointing [12]) trades additional computation for reduced memory usage.

and memory usage associated with attention key-value (KV) caches. RST focuses on the following configuration knobs, whose interactions collectively determine the memory-throughput trade-off.

(i) Prefix Caching [14]. LLM inference maintains a KV cache storing attention states for previously processed tokens. It exploits the observation that many requests share identical prompt prefixes (e.g., system prompts or few-shot examples) by caching their KV states, thereby avoiding redundant computation. When prefix caching is disabled, each request recomputes its full prompt context; when enabled, the inference engine retains KV states for reusable prefixes across requests, improving computation reuse at the cost of higher baseline GPU memory occupancy.

(ii) Continuous Batching. Modern LLM inference engines (e.g., vLLM [15]) support continuous batching, where requests dynamically join and leave an execution batch during decoding. Two parameters herein: the *maximum concurrent batch size* $B_{\max}$ and the *maximum sequence length* $L_{\max}$. Increasing $B_{\max}$ improves GPU utilization, while increasing $L_{\max}$ accommodates longer outputs; however, KV cache memory grows approximately as $B_{\max} \times L_{\max}$. The feasible ranges of both parameters depend on the model size and the available GPU memory budget.

(iii) GPU Memory Utilization Cap. Modern inference engines expose a *memory utilization cap* parameter (e.g., `gpu_memory_utilization`) that limits the fraction of physical GPU memory the engine may occupy. A higher cap allocates more memory to KV caches and enables deeper continuous batching, thereby increasing throughput; a lower cap leaves memory headroom for co-located jobs, facilitating GPU sharing at the cost of reduced per-job throughput.

3.1.2. Resource-Performance Profiling

For a given job j and configuration $c_j^i \in C_j$, RST constructs a resource-performance profile $p_j^i = \left(R_j(c),\ T_j(c),\ L_j\right)$.

(i) Resource Utilization Profiling, $R_j(c)$. The resource vector $R_j(c)$ characterizes the hardware resources required by configuration c_j^i. RST measures GPU SM utilization, GPU memory footprint, host CPU usage, and memory consumption via runtime interfaces such as NVIDIA DCGM.

(ii) Performance Profiling, $T_j(c)$. The performance metric $T_j(c)$ quantifies the execution efficiency of configuration c_j^i: training throughput (samples or iterations per second) for training workloads, and token-generation throughput or request completion rate for offline inference. RST additionally records auxiliary indicators such as iteration latency and GPU utilization, which are later used to estimate performance degradation under co-location.

Since deep learning training jobs typically run for hours or days, profiling the entire execution is impractical. Instead, ElastiCo collects runtime statistics for a small number of iterations after the warm-up phase, during which framework initialization, memory allocation, and data pipeline setup have stabilized. Training jobs exhibit highly stable per-iteration behavior after warm-up; therefore, these sampled iterations accurately represent steady-state performance.

(iii) Execution Constraints, L_j. The performance target L_j captures job-specific requirements that must be satisfied during scheduling, for example, minimum throughput targets for both training and offline inference workloads. These constraints prevent the scheduler from selecting configurations that violate application-level performance objectives.

When a profile is unavailable for a particular configuration, for example, due to a hardware change or a previously unseen knob combination, RST falls back to nearest-neighbor interpolation from the profile store, selecting the entry whose resource demand vector is closest under an ℓ_2 distance. This interpolation provides a provisional throughput estimate that is sufficient for initial scheduling decisions; once the job has been admitted, the online profiling procedure (§4.1) replaces the estimate with empirically measured values.

3.2. Elastic Shadow Pricing (ESP)

3.2.1. Problem Formulation

Given the profile families produced by RST, the scheduler must select a configuration $c_j \in C_j$ for every active job $j \in \mathcal{J}$ and decide which jobs to co-locate, so as to maximize aggregate weighted throughput subject to per-resource capacity and per-job performance constraints. We formalize this as the following integer program:

$$\max_{\{x_j^c\}} \sum_{j\in\mathcal{J}} \sum_{c\in C_j} w_j \cdot T_j(c) \cdot x_j^c \tag{1}$$

$$\text{s.t.} \quad \sum_{c\in C_j} x_j^c = 1, \quad \forall\, j \in \mathcal{J} \tag{2}$$

$$\sum_{j\in\mathcal{J}} \sum_{c\in C_j} R_j(c)[r] \cdot x_j^c \le C_r, \quad \forall\, r \in \mathcal{R} \tag{3}$$

$$T_j(c) \cdot x_j^c \geq L_j \cdot x_j^c, \quad \forall j,\, c \tag{4}$$

$$x_j^c \in \{0,1\}, \quad \forall j,\, c \tag{5}$$

where $x_j^c = 1$ if job j is assigned configuration c (and 0 otherwise), w_j is a per-job weight (e.g., inverse of remaining iterations for fairness), $T_j(c)$ and $R_j(c)$ are the throughput and resource vector from RST, C_r is the capacity of resource r, and L_j is the minimum throughput target for job j. Equation (2) ensures each job receives exactly one configuration; Equation (3) enforces cluster capacity; and Equation (4) enforces per-job performance.

This problem is NP-hard in general. The number of feasible solutions grows as $\prod_j |C_j|$, which is intractable for even moderate cluster sizes. Moreover, the formulation above does not yet account for co-location interference.

3.2.2. Lagrangian Decomposition

Rather than solving the integer program directly, ElastiCo relaxes the Equation (3) via Lagrange multipliers $\lambda = (\lambda_r)_{r\in\mathcal{R}}$, yielding the Lagrangian:

$$\mathcal{L}(\{x_j^c\}, \lambda) = \sum_j \sum_c \left[w_j T_j(c) - \textstyle\sum_r \lambda_r R_j(c)[r]\right] x_j^c + \sum_r \lambda_r C_r \tag{6}$$

For a fixed λ, the Lagrangian decomposes across jobs: each job j independently selects $c_j^* = \arg\max_c \left[w_j T_j(c) - \sum_r \lambda_r R_j(c)[r]\right]$. This per-job subproblem is trivially solvable by enumerating $|C_j|$ profiles (12–198 in our evaluation; see Table 2). The dual function $g(\lambda) = \max_{\{x_j^c\}} \mathcal{L}(\{x_j^c\}, \lambda)$ is then minimized over $\lambda \geq 0$ via projected subgradient ascent, which is the economic interpretation of the tâtonnement process described below.

3.2.3. Scoring and Price Update

The Lagrangian decomposition motivates a practical scoring mechanism. ElastiCo maintains a shadow-price vector λ^t for each resource dimension $r \in \mathcal{R}$. The price λ_r signals the scarcity of resource r across the cluster. In each scheduling epoch, every job $j \in \mathcal{J}$ independently selects its configuration c_j^* by minimizing a scoring function that corresponds to the negated per-job Lagrangian subproblem, augmented with interference and switching penalties:

$$\text{Score}_j(c; \lambda^t) = \text{PerformanceCost}_j(c) + \sum_{r\in\mathcal{R}} \lambda_r^t \cdot R_j(c)[r] + \Gamma_j^{(m)}(c), \tag{7}$$

Here, $\text{PerformanceCost}_j(c) = 1 - T_j(c)/T_j(c_j^{\max})$ measures the relative throughput loss of configuration c compared with the fastest configuration $c_j^{\max} = \arg\max_{c'} T_j(c')$, where $T_j(\cdot)$ is the throughput profiled by RST (§3.1). The term $\sum_{r\in\mathcal{R}} \lambda_r^t \cdot R_j(c)[r]$ is the resource rent, where each shadow price λ_r^t reflects the current scarcity of resource r. The penalty term $\Gamma_j^{(m)}(c)$ comprises two components: a *switching cost* $\gamma_0 \cdot \mathbb{1}[c \neq c_j^{\text{cur}}]$ that discourages unnecessary reconfiguration from the job's current configuration c_j^{cur}, and an *interference penalty* $\alpha \cdot \max(0,\ \hat{S}_{j|i} - \tau_j(t))$ injected by the IAC module (§3.3) when the predicted co-location slowdown $\hat{S}_{j|i}$ exceeds the adaptive tolerance threshold $\tau_j(t)$. The switching cost is evaluated at configuration-selection time using the job's current configuration; the interference penalty is updated after each ESP-IAC interaction round (Algorithm 1, lines 11–15).

Once all jobs have selected their configurations, the scheduler updates prices based on aggregate usage U_r relative to cluster capacity C_r:

$$\lambda_r^{t+1} = \max(0, \lambda_r^t + \eta_t(U_r - C_r)) \tag{8}$$

Here $U_r = \sum_{j\in\mathcal{J}} R_j(c_j^*)[r]$ is the aggregate demand for resource r under the current configuration selection, and $\eta_t > 0$ is the step size (learning rate), which may be fixed or follow a diminishing schedule $\eta_t = \eta_0/\sqrt{t}$ to balance responsiveness with stability.

This projected-subgradient update drives aggregate demand toward feasibility: if GPU memory is saturated, its price λ_{mem} rises, forcing jobs to select configurations with lower memory demand (e.g., enabling activation recomputation) until aggregate demand becomes feasible. Formally, the price update (Eq. 8) is a projected subgradient step on the Lagrangian dual of the capacity-constrained allocation problem (Eq. 6): the subgradient $U_r - C_r$ measures constraint violation, and projection onto $\mathbb{R}_{\geq 0}$ enforces non-negative prices.

3.3. Interference-Aware Co-location (IAC)

To increase GPU utilization, ElastiCo enables multiple workloads to run on the same GPU. Although such sharing boosts overall cluster use, co-located workloads can compete for shared hardware resources-streaming multiprocessors, memory bandwidth, GPU memory, and interconnects-leading to unpredictable slowdowns if not carefully controlled.

Rather than relying on bin-packing that only accounts for static capacities, ElastiCo performs *interference-aware co-location*: before placing two workloads on a GPU, the scheduler estimates the performance slowdown due to resource contention. If the projected degradation exceeds a tolerance threshold, it either avoids that placement or adjusts job settings. The IAC module predicts the slowdown factor $\hat{S}_{i|j}$ when jobs i and j run together.

3.3.1. Feature Engineering

The IAC prediction model takes as input a feature vector characterizing a candidate co-location pair and outputs the predicted performance decay. For each job pair (i, j), the model predicts $\hat{S}_{i|j} = T_i^{\text{solo}}/T_i^{\text{colo}}$, where T_i^{solo} denotes the throughput of job i running in isolation and T_i^{colo} denotes its throughput when co-located with job j. A value of $\hat{S}_{i|j} = 1$ indicates zero degradation; larger values indicate proportionally greater slowdown.

Hardware utilization features. Each workload configuration is profiled in isolation to capture its steady-state resource consumption. Two metrics characterize compute pressure: *SM Active*, the fraction of cycles with at least one active warp, and *SM Occupancy*, the ratio of resident to maximum warps per SM. Memory subsystem behavior is captured by *DRAM Active*, the fraction of cycles with active DRAM requests, and *Framebuffer Used*, the GPU memory occupied in megabytes. Finally, *Tensor Core Active* measures the fraction of cycles with active Tensor Core operations, distinguishing workloads dominated by matrix multiply-accumulate kernels from those that primarily execute general-purpose CUDA kernels.

Task-level features. All jobs share three common attributes: a binary workload type indicator (training or inference), the log-scaled model parameter count, and the isolated throughput $T_j(c)$ from RST profiling. Training jobs contribute three additional attributes: the micro-batch size (determining per-iteration activation memory and computation volume), an activation checkpointing flag, and a mixed-precision flag. Inference jobs contribute the input sequence length (ISL), the average output sequence length (OSL), and the prefix caching ratio ρ. ISL determines the computational cost of the prefill phase and the initial KV cache allocation; OSL determines decode duration and KV cache growth.

Pairwise interaction features. Beyond per-job features, the vector includes pairwise terms capturing the combined resource pressure. The mean of each hardware utilization metric across both jobs represents aggregate contention on the corresponding resource. A co-location type indicator encodes whether the pair is training–training, training–inference, or inference–inference, as these combinations exhibit distinct interference characteristics. A memory pressure ratio (sum of both jobs' framebuffer usage divided by total GPU memory capacity C_{mem} (e.g., 80 GB for A100)) quantifies proximity to the memory ceiling. A compute balance ratio $\min(sm_i, sm_j)/\max(sm_i, sm_j)$ measures the symmetry of SM demand. A memory intensity difference $|(dram_i/sm_i) - (dram_j/sm_j)|$ captures the complementarity of compute-bound and memory-bound profiles.

The complete feature vector has 36 dimensions: 14 per-job features for each job (5 hardware, 3 common, 3 training-specific, 3 inference-specific) and 8 pairwise features. Since co-located jobs generally experience different degrees of slowdown, each co-location experiment yields two training samples–one per job–with the per-job feature blocks swapped so that the first 14 dimensions always correspond to the job whose slowdown is being predicted.

3.3.2. Interference Prediction Engine

IAC employs a three-layer fully connected neural network to map the 36-dimensional feature vector $\mathbf{x}_{i,j}$ to the predicted slowdown $\hat{S}_{i|j}$. We adopt a DNN over tree-based alternatives for two reasons. First, the prediction target $\hat{S}_{i|j}$ is a continuous ratio that varies smoothly with resource utilization; a DNN with ReLU activations provides smooth interpolation in this space, whereas tree ensembles produce piece-wise constant predictions that can under-resolve small but scheduling-relevant slowdown differences (e.g., 1.05× vs. 1.15×). Second, the training set is moderately sized (constructed from sampled co-location experiments); Batch Normalization and Dropout (rate 0.2) regularize the network effectively on such data scales, while tree ensembles with comparable depth tend to overfit on correlated hardware-counter features.

The two hidden layers compute $h_\ell = \text{ReLU}(\text{BN}(W_\ell h_{\ell-1} + b_\ell))$ for $\ell \in \{1, 2\}$, each followed by dropout with rate 0.2. The output layer produces:

$$\hat{S}_{i|j} \;=\; 1 + \text{ReLU}\big(W_3 h_2 + b_3\big), \tag{9}$$

which constrains the prediction to $\hat{S}_{i|j} \geq 1.0$, consistent with the physical invariant that co-location cannot improve isolated performance. The model is trained offline by minimizing the log-scale mean squared error:

$$\mathcal{L} = \frac{1}{|D|} \sum_{(i,j)\in D} \left(\log \hat{S}_{i|j} - \log S_{i|j}\right)^2, \tag{10}$$

where $S_{i|j} = T_i^{\text{solo}}/T_i^{\text{colo}}$ is the ground-truth slowdown measured in offline co-location experiments. Operating in log space prevents samples with large slowdowns from dominating the gradient.

Training data generation. The training dataset is constructed by executing sampled workload pairs concurrently on a shared GPU via NVIDIA MPS. For each pair, both jobs are first profiled in isolation to obtain T_i^{solo} and the per-job feature vector; they are then co-located under identical configurations to measure T_i^{colo}. The ratio $S_{i|j} = T_i^{\text{solo}}/T_i^{\text{colo}}$ serves as the training label. In our evaluation setting, profiling 20–30% of all candidate pair combinations–stratified by workload type (training–training, training–inference, inference–inference) to ensure each co-location category is represented–empirically provides adequate coverage for the model to generalize to held-out pairs.

The predicted $\hat{S}_{i|j}$ feeds into the ESP scoring function (Eq. 7) through the interference penalty term $\Gamma_j^{(m)}(c)$, steering both configuration selection and GPU placement away from high-interference co-locations.

3.3.3. Adaptive Tolerance Thresholds

ElastiCo permits co-location only if the predicted slowdown for all participating jobs falls within a dynamic tolerance threshold $\tau_j(t)$. This threshold adapts to the cluster contention level: when the queue is long, ElastiCo tolerates slightly more interference to increase aggregate throughput.

$$\tau_j(t) = \tau_j^{\text{base}} + \beta \cdot \left(\frac{U(t)}{U_{\text{target}}} - 1\right) \tag{11}$$

where τ_j^{base} is the baseline tolerance, $U(t)$ is the current cluster utilization, U_{target} is the utilization target, and β controls sensitivity. The baseline τ_j^{base} is set to satisfy the job's throughput constraint: $\tau_j^{\text{base}} \leq T_j(c_j^*)/L_j$, ensuring that the admitted slowdown never causes the co-located throughput to drop below the minimum target L_j. The additive term $\beta \cdot (\cdot)$ is clamped so that $\tau_j(t) \leq \tau_j^{\text{base}} + \beta_{\max}$, providing a hard upper bound on tolerated degradation even under extreme cluster pressure.

3.4. Phase-Aware Disaggregated Scheduling (PDS)

RST exposes configuration flexibility, ESP determines optimal configurations under multi-resource constraints, and IAC quantifies interference risk. Individually, each module addresses one dimension of the co-location problem; without coordination, however, their decisions may conflict–e.g., ESP may select a memory-intensive configuration that IAC subsequently flags as interference-prone, triggering wasteful rollbacks. The Phase-Aware Disaggregated Scheduling (PDS) module closes this loop by integrating the three modules into a unified, iterative scheduling cycle that jointly converges on resource allocations that are simultaneously capacity-feasible, interference-safe, and performance-compliant.

Specifically, the predicted slowdown factors from IAC are incorporated directly into the configuration-selection process by penalizing profiles that introduce excessive interference. For a candidate co-location pair (i, j), if the predicted slowdown $\hat{S}_{i|j}$ exceeds the tolerance threshold for either workload, the scheduler reduces the effective value of the offending configuration through the penalty term in the ESP objective (Eq. 7), encouraging the tâtonnement process to select alternative configurations or placements with lower interference.

Through this feedback loop, ElastiCo balances two competing goals: maximizing GPU utilization through aggressive co-location and preserving application performance by avoiding harmful contention. Rather than rejecting co-location decisions outright, the system gradually steers allocations toward interference-safe configurations.

Scheduling Workflow. Algorithm 1 summarizes the complete scheduling loop. The scheduler operates periodically at a fixed epoch interval (default $\Delta T = 5$ s), enabling rapid reaction to workload arrivals, job completions, and configuration changes.

At the beginning of each epoch, the scheduler computes the *effective resource capacity* by reserving headroom for queued jobs:

$$\mathbf{C}_{\text{eff}}(t) \;=\; \mathbf{C} \;-\; \gamma \sum_{j \in \mathcal{Q}(t)} R_j^{\min}, \tag{12}$$

where $\mathbf{C}$ is the physical cluster capacity, $\mathcal{Q}(t)$ is the set of pending jobs, $R_j^{\min} = \min_{c \in C_j} R_j(c)$ is the smallest resource footprint in job j's profile family (from RST), and $\gamma \in [0, 1]$ controls reservation aggressiveness (γ=0 ignores the queue; γ=1 reserves capacity for every pending job at its minimum footprint). By operating on $\mathbf{C}_{\text{eff}}$ rather than $\mathbf{C}$, ESP avoids over-committing resources needed for imminent job admissions, reducing reactive migrations.

Algorithm 1 ElastiCo Scheduling Loop

Require: Active jobs $\mathcal{J}$, queue $\mathcal{Q}$, profile sets $\{\mathcal{P}_j\}_{j \in \mathcal{J}}$, capacity $\mathbf{C}$, prices λ, max rounds M
Ensure: Configuration selection $\{c_j^*\}$, placement $\mathcal{A}$, updated prices λ

$\mathbf{C}_{\text{eff}} \leftarrow \mathbf{C} - \gamma \sum_{j \in \mathcal{Q}} R_j^{\min}$ {Reserve capacity for queued jobs}
$\mathbf{\Gamma} \leftarrow \mathbf{0}$ {Initialize interference penalties}
for $m = 1, \ldots, M$ **do**
  $\lambda^* \leftarrow \textsc{EspTatonnement}(\mathcal{J}, \{\mathcal{P}_j\}, \mathbf{C}_{\text{eff}}, \lambda, \mathbf{\Gamma})$
  for each job $j \in \mathcal{J}$ **do**
    $c_j^* \leftarrow \arg\min_{c \in C_j} \textsc{Score}_j(c;\ \lambda^*)$
  end for
  $\mathcal{A} \leftarrow \textsc{BinPack}\big(\{(j,\ R_j(c_j^*))\},\ \text{GPU nodes}\big)$
  $\mathcal{V} \leftarrow \emptyset$ {Violation set}
  for each GPU g with co-located pair $(i, j) \in \mathcal{A}$ **do**
    $\hat{S}_{i|j},\ \hat{S}_{j|i} \leftarrow \textsc{IacPredict}(i, j)$
    if $\hat{S}_{i|j} > \tau_i(t)$ **or** $\hat{S}_{j|i} > \tau_j(t)$ **then**
      Update $\mathbf{\Gamma}$ for offending configurations
      $\mathcal{V} \leftarrow \mathcal{V} \cup \{(i, j)\}$
    end if
  end for
  if $\mathcal{V} = \emptyset$ **then**
    break {All co-location pairs safe}
  end if
end for
Apply configuration switches, placements, and migrations
Admit jobs from $\mathcal{Q}$ if residual capacity permits
return $\{c_j^*\}_{j \in \mathcal{J}},\ \mathcal{A},\ \lambda^*$

The scheduling loop proceeds as follows. The ESP module first computes shadow prices through a tâtonnement process based on current resource demand and capacity constraints. Given the resulting price vector λ^*, each job independently selects the configuration that maximizes its net surplus–the gap between configuration value and priced resource consumption.

A bin-packing procedure then assigns jobs to GPU devices according to their resource demands. Because bin-packing considers only static capacity constraints, the resulting placement may still introduce runtime interference. The IAC module therefore evaluates every co-located pair using the interference prediction engine.

If the predicted slowdown for any pair exceeds the adaptive threshold $\tau_j(t)$, the scheduler updates the corresponding configuration penalties and records the violation. These penalties reduce the attractiveness of problematic configurations in subsequent ESP iterations. The loop repeats until no violations remain or the maximum number of rounds M is reached.

Runtime Adaptation. After convergence, the scheduler applies the resulting allocation decisions, which may involve switching job configurations (e.g., changing batch size or checkpointing depth), migrating workloads across

GPUs, or admitting queued jobs when capacity becomes available. Because in-place configuration adjustments–propagated through Pod specification parameters (§4.2)–avoid the overhead of cross-node state transfer, PDS prioritizes reconfiguration over migration whenever possible.

Computational Complexity. The per-epoch complexity of the scheduling loop is $O\big(M \cdot (I \cdot N \cdot \bar{K} + N^2 \cdot F)\big)$, where M is the maximum number of ESP–IAC interaction rounds (typically 2–3), I is the number of tâtonnement iterations for price convergence (typically 3–5 with warm starting), N is the number of active jobs, $\bar{K}$ is the average number of profiles per job, and F is the cost of a single interference prediction.

In practice, each interference prediction takes approximately 0.08 ms. Even for large-scale scenarios with $N = 500$ active jobs and $\bar{K} = 8$ profiles per job, the end-to-end scheduling latency remains under 15 ms–negligible relative to the 5-second epoch interval–enabling ElastiCo to operate efficiently at cluster scale while continuously adapting to dynamic workload conditions.

4. Implementation

Translating the design of §3 into a production-grade system raises two principal engineering challenges that are not addressed by the algorithmic formulation alone: (i) *non-intrusive observability*–collecting the hardware-counter and kernel-level telemetry required by RST and IAC; and (ii) *memory isolation under co-location*–enforcing per-job GPU memory budgets when multiple workloads share a device via NVIDIA MPS. Configuration changes selected by ESP are propagated to workloads through Pod specification parameters and, when necessary, through checkpoint-based state serialization (§4.2), ensuring safe transitions without requiring modifications to user code.

4.1. Non-Intrusive Interception and Profiling

ElastiCo measures application resource usage and builds RST profiles non-intrusively, ensuring compatibility with arbitrary PyTorch, vLLM, and custom CUDA applications. This is achieved through a *Two-Dimensional Optimized Profiler* (TDOP) that fuses two complementary telemetry streams: (i) a *software dimension* that intercepts CUDA driver and runtime library invocations at the user-space level to capture per-kernel launch metadata and memory allocation events, and (ii) a *hardware dimension* that continuously samples GPU performance counters via NVIDIA DCGM to characterize steady-state compute and memory subsystem behavior. The fusion of these two streams–software-level allocation traces and hardware-counter telemetry–provides richer and more accurate profiles than either stream alone, because software events reveal *what* the application requests while hardware counters reveal *how* the GPU responds.

Instrumentation mechanism. The TDOP intercepts CUDA entry points via `LD_PRELOAD`-based dynamic library interposition, inserting lightweight wrappers around `cuLaunchKernel`, `cudaMalloc`, `cudaFree`, and collective communication primitives. The wrappers record per-kernel launch metadata (grid/block dimensions, shared memory size, stream ID) and memory allocation events without modifying or delaying the underlying calls. Aggregated metrics are written to a per-process ring buffer and flushed to the data-plane agent at configurable intervals (default: every 2 seconds).

Profiling. For each new (job type, configuration) pair, the profiling procedure proceeds in five steps:

(i) Launch. The job is started under the target configuration in an isolated profiling context (a dedicated Kubernetes pod with MPS disabled to prevent interference).

(ii) Warm-up. The job runs for a fixed warm-up budget (default: 50–100 iterations, or approximately 30 seconds, whichever comes first) to allow framework initialization, JIT compilation, memory allocator warm-up, and data pipeline prefetching to stabilize.

(iii) Measurement. After warm-up, the profiler collects telemetry for a measurement window of approximately 60 seconds, sampling hardware counters at 1-second resolution and recording per-iteration wall-clock time and throughput. Transient outliers (e.g., from garbage collection or checkpoint I/O) are removed using a 10%–90% trimmed mean.

(iv) Profile construction. The profiler computes steady-state estimates of $T_j(c)$ (throughput) and $R_j(c)$ (multi-dimensional resource vector) and derives the 36-dimensional IAC feature vector from the hardware-counter and task-level metadata. All values are written to the profile store and tagged with the current hardware signature.

(v) Stop. The profiling pod is terminated.

Amortization and caching. Profile generation takes 3–5 minutes per (model architecture, configuration) pair, but this cost is amortized across all jobs of the same type via the profile store. The cache hit rate in practice is high

(approximately 94% of job arrivals match an existing profile) because the set of model architectures deployed in a cluster stabilizes quickly. When a hardware upgrade or framework version change is detected via the hardware signature, only the affected profiles are invalidated, avoiding a full re-profiling cycle.

For unseen configurations or hardware changes, the system falls back to nearest-neighbor interpolation as described in §3.1; empirically measured profiles take precedence once available.

4.2. Configuration Propagation and Memory Budget Coordination

ElastiCo is a scheduler-level system: it decides *what* configuration each job should run under and *where* the job should be placed, but it does not replace or bypass the memory allocators provided by deep learning frameworks or the GPU driver. All resource budgets are enforced by setting the appropriate launch parameters in the Pod specification before the job starts (or restarts after a heavy reconfiguration). This section describes how those parameters are chosen and propagated.

Memory budget derivation. For each job j assigned to configuration c_j^* by ESP, the scheduler computes a memory budget $M_j = R_j(c_j^*)[\texttt{GPU\_mem}]$ from the RST profile. When two jobs j_1, j_2 are co-located on the same GPU by PDS, the scheduler verifies the packing constraint $M_{j_1} + M_{j_2} \leq M_{\text{GPU}} - M_{\text{sys}}$, where M_{GPU} is the total device memory and M_{sys} is a reserved margin for driver and MPS overhead (default: 512 MB).

Framework-specific parameter mapping. The computed memory budget is translated into framework-native parameters and injected into the Pod specification as environment variables or container arguments:

(i) vLLM inference. The budget is mapped to `gpu_memory_utilization` $= M_j / M_{\text{GPU}}$, which controls the fraction of device memory that the vLLM engine pre-allocates for KV-cache blocks and model weights.

(ii) PyTorch training. The budget is mapped to `PYTORCH_CUDA_ALLOC_CONF=max_split_size_mb:`M_j and, where supported, to `torch.cuda.set_per_process_memory_fraction(`M_j / M_{GPU}`)`, which caps the CUDA caching allocator's consumption.

(iii) MPS resource limits. When NVIDIA MPS is enabled on the node (a cluster-level infrastructure decision outside ElastiCo's scope), the `CUDA_MPS_PINNED_DEVICE_MEM_LIMIT` environment variable is set per client process to enforce a hard memory ceiling at the driver level.

Because each framework's allocator operates within the externally imposed ceiling, co-located jobs are isolated without requiring a custom GPU memory manager. This *budget-from-above* design leverages the memory-capping interfaces that frameworks already expose, translating scheduler-computed budgets into framework-native parameters. This preserves the non-intrusiveness guarantee (§4) while achieving precise memory isolation under co-location.

State serialization for migration. When a job must be migrated–e.g., because IAC identifies a harmful co-location pair at runtime–ElastiCo triggers a framework-native checkpoint through the data-plane sidecar agent. For PyTorch training jobs this invokes `torch.save()` on model parameters and optimizer states; for vLLM inference services the engine is gracefully drained and the model weights are already resident in host memory or can be reloaded from the model repository. The checkpoint is written to a shared persistent volume (NVMe-backed by default) accessible from any node, so the replacement pod can restore without data movement across the network. Serialization runs asynchronously: the scheduling epoch is blocked only for the brief final consistency flush (typically < 1 s for a 7 B-parameter model).

5. Evaluation

We evaluate ElastiCo through testbed experiments on a 64-GPU cluster and large-scale trace-driven simulations (up to 512 GPUs), addressing six questions:

- **Q1** How does ElastiCo compare to state-of-the-art baselines in JCT, throughput, and utilization? (§5.2)
- **Q2** How does each component (RST, ESP, IAC, PDS) contribute to overall performance? (§5.3)
- **Q3** What scheduling behaviors produce the observed gains? (§5.4)
- **Q4** How accurate is the IAC interference predictor? (§5.5)
- **Q5** How does ElastiCo scale with cluster size and load intensity? (§5.6)
- **Q6** What system overheads does ElastiCo incur? (§5.7)

5.1. Experimental Setup

Testbed. 8 nodes, each with 8 NVIDIA A100-40GB GPUs (64 GPUs total), interconnected via 200 Gbps InfiniBand HDR with intra-node NVSwitch. All nodes run Ubuntu 22.04 with CUDA 12.8, PyTorch 2.8, and vLLM 0.11.0.

Table 2
Workloads used in evaluation. Each workload is profiled under multiple configurations by varying the listed knobs; the resulting Cartesian product defines the set of feasible profiles per workload.

Workload	Type	Mem. (GB)	Configuration Knobs
ResNet-50 [16]	Train	12–24	BS∈{32,64, ⋯,512}, CK, AMP
BERT [17]	Train	24–36	BS∈{32,64,128}, CK, AMP
DCGAN [18]	Train	8–16	BS∈{32,64,⋯,512}, CK, AMP
PointNet [19]	Train	12–20	BS∈{32,64,⋯,512}, CK, AMP
Transformer [20]	Train	20–38	BS∈{32,64,⋯,512}, CK, AMP
LSTM [21]	Train	4–12	BS∈{32,64,⋯,4096}, CK, AMP
NCF-NeuMF [22]	Train	4–16	BS∈{32,64,⋯,8192}, CK, AMP
PPO [23]	Train	8–24	BS∈{32,64,⋯,4096}, CK, AMP
DeepSeek-1.5b [24]	Infer	16–36	BS∈{1,2,4}, SL∈{4K,8K,16K}, PC, MU 0.4–0.9
Qwen-1.7b [25]	Infer	16–36	BS∈{1,2,4}, SL∈{4K,8K,16K}, PC, MU 0.4–0.9
Mistral-7b [26]	Infer	28–36	BS∈{1,2,4}, SL∈{1K,2K,4K}, PC, MU 0.7–0.9
LLaMA-3-7b [27]	Infer	28–36	BS∈{1,2,4}, SL∈{1K,2K,4K}, PC, MU 0.7–0.9

BS = Batch Size; SL = Sequence Length; CK = Activation Checkpointing (±); AMP = Automatic Mixed Precision (±); PC = Prefix Caching (±); MU = `gpu_memory_utilization` (vLLM), sampled at 0.05 intervals. ± denotes an on/off toggle; Knob values form a Cartesian product of all listed options.

Workloads. Table 2 lists the 12 workloads used in our evaluation, spanning eight training tasks and four offline inference tasks. Training workloads cover diverse model architectures: image classification (ResNet-50), language understanding (BERT), generative modeling (DCGAN), 3D point cloud processing (PointNet), sequence modeling (Transformer, LSTM), recommendation (NCF-NeuMF), and reinforcement learning (PPO). Inference workloads target four LLMs of varying scale: DeepSeek-1.5b, Qwen-1.7b, Mistral-7b, and LLaMA-3-7b. Each workload is profiled under multiple configurations by varying batch size, activation checkpointing, mixed-precision training, sequence length, and prefix caching, yielding diverse feasible profiles per workload as detailed in the table. All workloads are executed on the physical testbed described above.

Simulation. For large-scale experiments beyond the 64-GPU testbed, we use a discrete-event simulator calibrated against testbed measurements. The simulator models GPU resource contention, MPS overhead, and interference using the IAC predictor. We validate the simulator by replaying the 24-hour testbed trace at 64 GPUs and comparing JCT and throughput: simulated results match testbed measurements to within 7%, confirming fidelity. The simulator is used for scalability experiments at 128, 256, and 512 GPUs (§5.6).

Baselines. We compare against two representative schedulers: **Volcano** [28], a widely-adopted Kubernetes-native batch scheduler that provides gang scheduling, queue management, and fair-share policies but treats job resource requests as fixed and does not perform co-location optimization; and **Lucid** [10], a state-of-the-art DL training scheduler that uses lightweight profiling, indolent resource packing, and priority-based scheduling to minimize average JCT without modifying training code. Volcano represents production-grade static scheduling, while Lucid represents the best available elastic training scheduler.

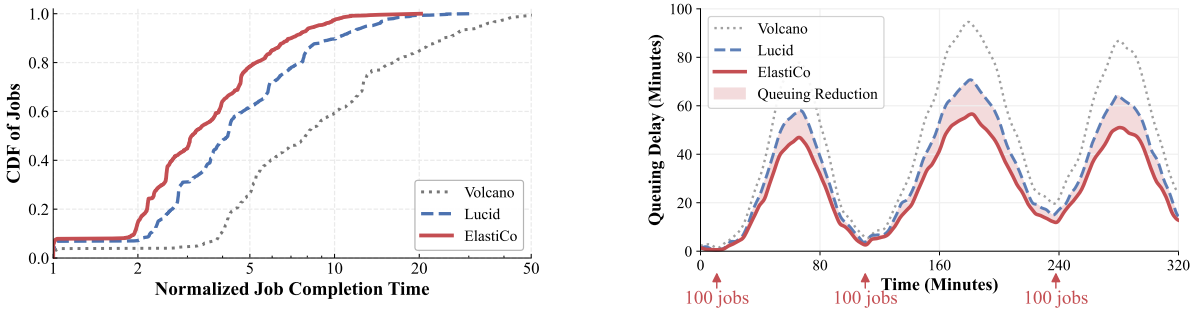
Baseline justification. Table 3 positions ElastiCo against recent co-location systems. SIRIUS [6], Mudi [1], and GPUColo [8] are the closest co-location-aware comparators; however, none have publicly available implementations or standardized benchmark suites, and each targets a different workload mix (SIRIUS: online inference priority; Mudi: SLO-constrained inference batching; GPUColo: latency-guaranteed GPU-local sharing). To provide the fairest comparison possible, we select Volcano and Lucid–whose open-source implementations are mature and reproducible–as anchors representing the two ends of the scheduling spectrum (static vs. elastic), and include qualitative feature comparison against co-location systems below. We encourage future work to establish standardized co-location benchmarks.

Metrics. (1) Average JCT (job completion time, submission to finish); (2) Cluster throughput (total useful work per unit time, normalized); (3) GPU SM utilization; (4) Throughput attainment (fraction of inference tasks meeting throughput targets; fraction of training jobs meeting throughput targets); (5) GPU instance count required for a given workload mix.

Table 3
Qualitative comparison with representative schedulers. "Config. reshape" denotes whether the scheduler modifies intra-job configuration knobs (batch size, checkpointing, precision); "Interf. pred." denotes whether interference is predicted before co-location; "Cluster-wide" denotes cluster-level multi-resource optimization (vs. GPU-local).

System	Co-location.	Config. reshape	Interf. pred	Cluster-wide	Train+Infer	Open source
Volcano [28]	×	×	×	✓	×	✓
Lucid [10]	×	Partial	×	✓	×	✓
SIRIUS [6]	✓	×	×	×	✓	×
Mudi [1]	✓	×	✓	×	✓	×
GPUColo [8]	✓	×	×	×	✓	×
SMore [7]	✓	×	✓	×	Partial	×
ElastiCo	✓	✓	✓	✓	✓	–

"Partial" for Lucid: adjusts GPU count but not intra-job knobs. "Partial" for SMore: manages training throughput but not inference throughput.



(a) CDF of normalized JCT. ElastiCo achieves 2.5× lower median JCT than Volcano.

(b) Per-job queuing delay over time. ElastiCo significantly reduces queuing delay and prevents backlog accumulation throughout the workload arrival period.

Figure 6: Job-level performance comparison on the 64-GPU testbed under mixed workload.

5.2. Overall Performance

Figure 6a shows the CDF of normalized JCT for training jobs. ElastiCo reduces average JCT by 2.94× compared to Volcano and by roughly 1.35× compared to Lucid, while achieving approximately 2.5× lower median JCT than Volcano. The improvement is driven by two factors: (i) RST enables training jobs to start on fewer GPUs rather than waiting for full allocation, reducing queuing time; (ii) ESP dynamically reconfigures jobs as resources become available, recovering throughput without user intervention.

Figure 6b shows per-job queuing delay as a function of arrival order. Volcano exhibits rapidly growing queuing delays as the cluster fills, while Lucid mitigates this partially through elastic packing. ElastiCo significantly reduces queuing delay and mitigates backlog growth even for late-arriving jobs.

Figure 7 shows GPU utilization over a 24-hour trace replay. ElastiCo raises average SM utilization from 25% (Volcano) to 46%, corresponding to a 1.84× improvement, while Lucid reaches 38%. The utilization gain is sustained throughout the day: during off-peak hours, training jobs expand to use GPUs vacated by reduced inference demand; during peak hours, RST compresses training configurations to free capacity for inference.

Table 4 summarizes the aggregate comparison across the main metrics. Relative to Volcano, ElastiCo reduces average JCT to 0.34× and improves cluster throughput to 2.02×, while raising average GPU SM utilization from 25% to 46%. Compared to Lucid, ElastiCo further improves both JCT and throughput, indicating that configuration-aware reshaping and interference-aware co-location provide benefits beyond elastic packing alone.

ElastiCo also reduces the number of GPU instances required to serve a given mix of training and inference workloads. By co-locating complementary workloads and reshaping configurations to fit available capacity, ElastiCo achieves a 44% reduction in additional GPU instances compared to static partitioning–equivalent to serving the same aggregate workload on substantially fewer devices.

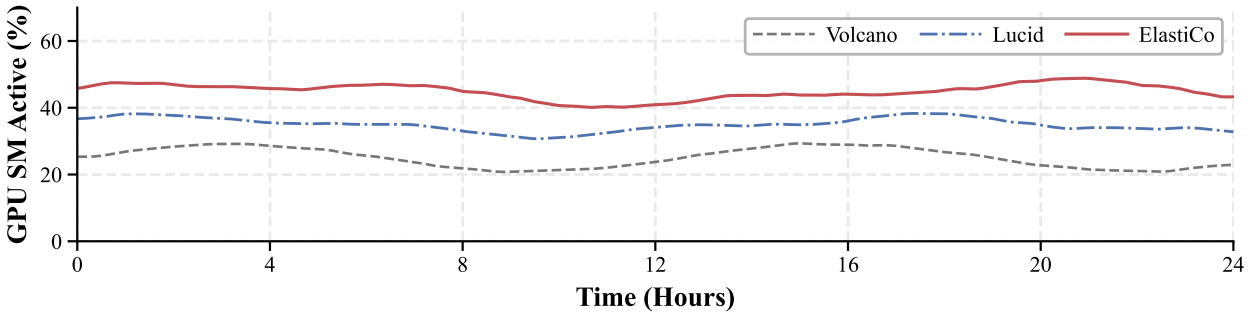


Figure 7: Cluster-wide GPU SM Active over a 24-hour trace replay. ElastiCo sustains around 40–50% utilization, compared to roughly 20–30% for Volcano.

Table 4
Main comparison results (64-GPU testbed, 24-hour mixed workload). Avg JCT and throughput are normalized to Volcano. Attainment measures the fraction of jobs meeting their throughput target.

System	Avg JCT	Throughput	SM Util. (%)	Attainment (%)
Volcano	1.00×	1.00×	25	100
Lucid	0.46×	1.76×	38	100
ElastiCo	**0.34×**	**2.02×**	**46**	**98.3**

Volcano and Lucid achieve 100% attainment trivially by running jobs in isolation (no co-location interference). ElastiCo's 98.3% attainment reflects that IAC successfully prevents harmful co-locations, with the 1.7% shortfall arising from transient interference during configuration transitions.

Table 5
Ablation study: impact of disabling each ElastiCo component.

Configuration	Avg JCT	Throughput	Mem Util.%	SM Util.%
Full ElastiCo	**1.00×**	**1.00×**	**82.6**	**46.0**
w/o RST	1.53×	0.77×	73.1	35.4
w/o ESP	1.25×	0.82×	77.4	38.9
w/o IAC	1.08×	0.92×	80.8	43.1
w/o PDS	1.17×	0.89×	79.2	40.6

5.3. Ablation Study

To isolate each module's contribution, we disable one component at a time while keeping the rest intact. When RST is disabled, every job runs under a fixed default configuration; when ESP is disabled, a greedy first-fit policy replaces the pricing mechanism; when IAC is disabled, co-location is admitted without interference checking; when PDS is disabled, scheduling decisions are applied without queue-aware capacity reservation or coordinated reconfiguration. Table 5 summarizes the results.

RST Unlocks the Co-location Space. Disabling RST causes the largest degradation: JCT increases by 53% and throughput drops by 23%. The reason is visible in the interference data (Table 6): many workload pairs fall into a moderate-conflict regime (decay $\approx$ 0.6–0.9) that is infeasible under rigid configurations but becomes feasible once RST reshapes resource demands. For example, enabling activation checkpointing for a BERT job reduces its memory footprint from 36 GB to 24 GB, opening sufficient headroom for co-location with an inference task. GPU memory utilization drops from 82.6% to 73.1% without RST, confirming that fixed configurations leave substantial capacity stranded. **Insight:** The dominant source of improvement is demand reshaping, not job reordering.

IAC Prevents Catastrophic Interference. Removing IAC only slightly reduces throughput (about 8%) but leads to noticeably worse co-location quality, because high-conflict pairs are admitted without restriction and can cause

Table 6
Representative co-location pairs and observed interference. Decay denotes the fraction of solo throughput retained under co-location (1.0 = no degradation).

Severity	Workload Pair (A / B)	Decay_A	Decay_B	Type
Severe	Qwen-1.7b^{PC} / ResNet-50^{CK}	0.48	0.55	I + T
	BERT° / BERT^{CK}	0.47	0.57	T + T
	DeepSeek-1.5b^{PC} / ResNet-50^{CK}	0.50	0.56	I + T
High	Qwen-1.7b^{PC} / ResNet-50°	0.52	0.57	I + T
	PPO° / ResNet-50°	0.54	0.72	T + T
	Qwen-1.7b° / ResNet-50^{CK}	0.66	0.57	I + T
	DeepSeek-1.5b° / ResNet-50^{CK}	0.67	0.61	I + T
Moderate	Qwen-1.7b° / ResNet-50°	0.71	0.68	I + T
	DeepSeek-1.5b° / PPO°	0.56	0.89	I + T
	LSTM° / ResNet-50^{CK}	0.70	0.70	T + T
	LSTM° / NCF°	0.75	0.55	T + T
	Qwen-1.7b^{PC} / DeepSeek-1.5b^{PC}	0.60	0.62	I + I
	LLaMA-3-7b^{PC} / Mistral-7b^{PC}	0.58	0.61	I + I
Low	Qwen-1.7b° / NCF°	0.93	0.95	I + T
	BERT^{CK} / DeepSeek-1.5b°	0.95	0.96	T + I
Near-zero	NCF° / PPO°	0.97	0.98	T + T
	BERT° / PPO°	0.98	0.98	T + T
	NCF° / BERT°	0.99	0.99	T + T

T = Training, I = Inference. ° = Default configuration; PC = PrefixCaching; CK = Activation Checkpointing.

severe slowdown for individual jobs. **Insight:** IAC is essential for avoiding localized performance collapse rather than improving average throughput.

ESP Improves Global Allocation Efficiency. Replacing ESP with a greedy policy reduces throughput by 18%. Interference is highly heterogeneous (Table 6), with some pairs nearly interference-free and others significantly degraded. Greedy decisions fail to account for this variability, while ESP prioritizes globally efficient, low-interference combinations. **Insight:** Exploiting heterogeneity requires global coordination.

PDS as the Orchestration Layer. PDS integrates configuration selection, interference feedback, and runtime adaptation into a unified loop. Disabling it leads to moderate degradation, as decisions become unstable under dynamic workloads. Without coordinated handling of reconfiguration and interference feedback, the system suffers from inconsistent placement and unnecessary adjustments. **Insight:** Effective scheduling requires not only good decisions, but consistent execution.

5.4. Case Study

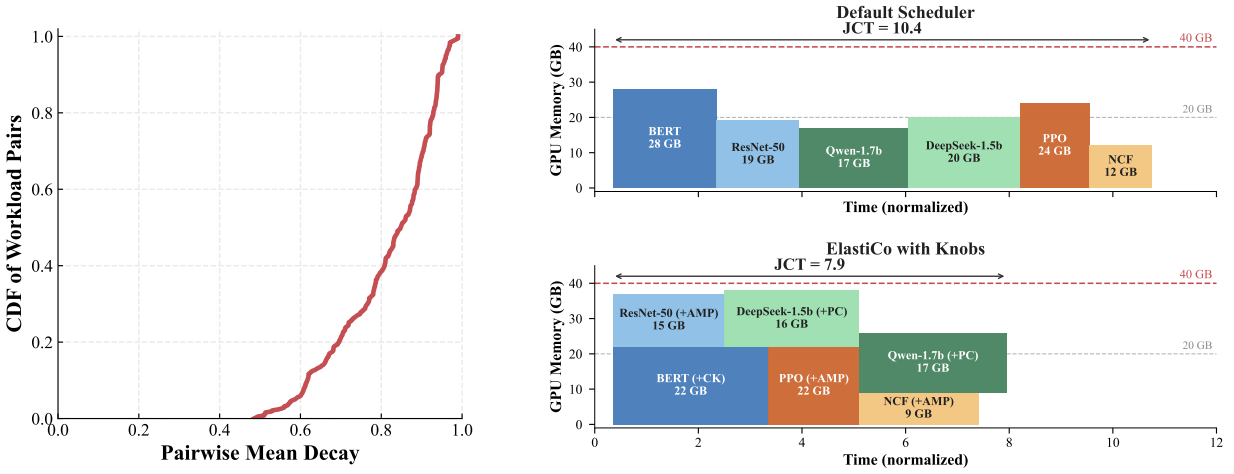
To better understand where the performance gains of ElastiCo come from, we combine the global interference structure with concrete co-location examples. Figure 8(a) characterizes the overall distribution of pairwise interference, while Figure 8(b) presents representative execution timelines for selected workload pairs.

Interference structure.

As shown in Figure 9, co-location interference exhibits strong heterogeneity: the observed performance decay ranges from near-zero impact to significant slowdown. Moreover, interference is not solely determined by workload type, but depends critically on their configuration-dependent resource demands, including memory footprint and compute intensity. This suggests that co-location feasibility and efficiency are fundamentally shaped by how resource demands align under specific configurations.

Motivated by this observation, we analyze three representative scenarios that illustrate distinct mechanisms through which ElastiCo improves scheduling quality.

Case 1: Infeasible co-location resolved by RST reshaping. Under the default scheduler, BERT (36 GB) and ResNet-50 (24 GB) each run with their peak-throughput configuration, totaling 60 GB–well beyond the 40 GB capacity of a



(a) CDF of pairwise mean performance decay across workload pairs.

(b) Resource demand timelines under co-location for representative workload pairs.

Figure 8: Interference characteristics of workload co-location.

single A100. The scheduler serializes them, leaving one GPU idle while the other runs. ElastiCo enables CK for BERT (reducing its footprint to 24 GB) and AMP for ResNet-50 (reducing to 14 GB). The combined 38 GB fits within device capacity, enabling concurrent execution and reducing the pair's makespan by 41%. **Insight:** rigid configurations create hard infeasibility; RST converts infeasible pairs into feasible co-locations by reshaping resource demands.

Case 2: Avoidable queuing resolved by ESP pricing. Consider a burst of four training jobs (ResNet-50, BERT, PPO, NCF) arriving simultaneously on a two-GPU node. Under a greedy first-fit policy, the scheduler assigns the two largest jobs (BERT, ResNet-50) to separate GPUs and queues PPO and NCF. ESP, by contrast, raises GPU memory prices in response to the overload, steering all four jobs toward lower-memory configurations (smaller batch sizes, AMP enabled). This allows all four jobs to start immediately via two co-located pairs, eliminating the queuing delay entirely. The 30-minute queuing time observed under the greedy policy is reduced to zero, at a cost of 8–12% per-job throughput reduction from the configuration downgrade–a trade-off that reduces average JCT by 35%. **Insight:** ESP's price signals dynamically steer jobs toward configurations that collectively fit, converting queuing delays into controlled throughput trade-offs.

Case 3: Configuration-enabled dense packing. Figure 8(b) compares the execution timelines of six representative workloads under the default scheduler and ElastiCo. Under the baseline scheduler, all jobs run with fixed configurations and rigid resource allocations. Due to their high GPU memory footprints, the combined demand of multiple jobs exceeds device capacity, making concurrent execution infeasible. As a result, the scheduler is forced to execute jobs in a largely serialized manner, with minimal overlap dictated by residual capacity rather than scheduling decisions. This leads to significant idle periods and an overall job completion time (JCT) of 10.4. ElastiCo overcomes this limitation by actively reshaping workload resource demands through configuration knobs. Specifically, checkpointing (CK) is enabled for BERT, AMP is applied to ResNet-50, PPO, and NCF, and prefix caching (PC) is enabled for the inference workloads (Qwen-1.7b and DeepSeek-1.5b). These transformations reduce per-job memory footprints, allowing the aggregate demand to fit within device capacity. Multiple jobs become jointly schedulable, enabling substantial temporal overlap. The overall JCT is reduced from 10.4 to 7.9–a 24% improvement from configuration-level resource reshaping alone. **Insight:** the combination of RST reshaping, ESP pricing, and IAC interference checking produces a densely packed schedule that no single mechanism can achieve in isolation.

5.5. Interference Prediction Accuracy

The IAC module's utility depends on the accuracy of its slowdown predictions. We evaluate the DNN-based predictor using 5-fold cross-validation on the co-location dataset collected from the testbed, which comprises pairwise measurements across all workload combinations listed in Table 2 under multiple configuration variants.

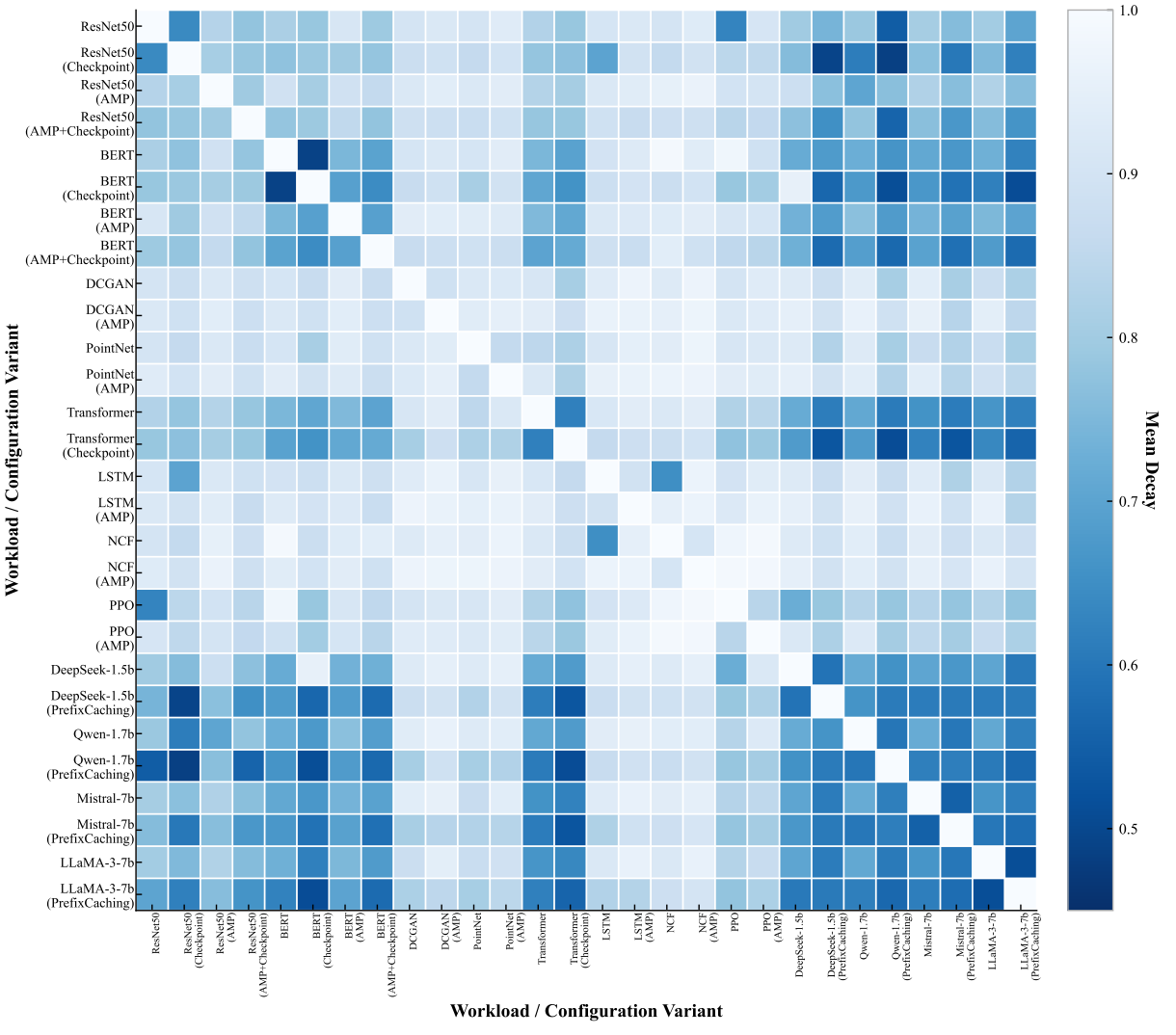


Figure 9: Heatmap of pairwise co-location interference across workload variants.

Table 7 reports per-category and overall accuracy. The predictor achieves a mean absolute percentage error (MAPE) of 7.6% across all co-location types, with an R^2 of 0.89. Training–training pairs are predicted most accurately (6.8% MAPE), likely because training workloads exhibit more stable, periodic resource consumption. Training–inference pairs show slightly higher error (8.2%), reflecting the greater variability introduced by continuous-batching dynamics in inference workloads. Importantly, prediction errors are small relative to the interference magnitudes observed in practice (Table 6): the difference between a safe pair (decay $>$ 0.9) and a harmful pair (decay $<$ 0.6) spans 30–40 percentage points, while the predictor's typical error is under 8 percentage points. This margin is sufficient for the scheduler to reliably distinguish safe from harmful co-locations.

We additionally compare the DNN predictor against two alternatives: a gradient-boosted decision tree (XGBoost) and a linear regression baseline. The DNN achieves 7.6% MAPE, compared to 9.4% for XGBoost and 18.7% for linear regression. The DNN's advantage is most pronounced for moderate-interference pairs (decay 0.6–0.9), where smooth interpolation in the feature space is critical for scheduling-relevant distinctions–consistent with the design rationale in §3.3.

Table 7
IAC prediction accuracy (5-fold cross-validation). MAPE = mean absolute percentage error; R^2 = coefficient of determination.

Co-location Type	MAPE (%)	R^2	Samples
Training + Training	6.8	0.91	384
Training + Inference	8.2	0.88	512
Inference + Inference	7.5	0.89	192
Overall	**7.6**	**0.89**	**1088**

Table 8
Scalability: ElastiCo performance at increasing cluster scale (simulation). All metrics normalized to the Volcano baseline at the same scale.

GPUs	Avg JCT	Throughput	SM Util. (%)	Sched. Latency (ms)
64	0.34×	2.02×	46	4.2
128	0.36×	1.95×	44	6.8
256	0.38×	1.89×	43	9.1
512	0.41×	1.82×	41	13.7

5.6. Scalability

To assess ElastiCo's behavior beyond the 64-GPU testbed, we use the discrete-event simulator described in §5.1 to evaluate performance at 64, 128, 256, and 512 GPUs. The simulator is calibrated against testbed measurements and validated to within 7% of physical-cluster results for JCT and throughput metrics. Workload arrival rates are scaled proportionally to cluster size to maintain a consistent load factor.

Table 8 shows that ElastiCo maintains substantial improvements over the baseline across all scales. At 512 GPUs, ElastiCo still reduces average JCT by 2.44× and improves throughput by 1.82× relative to Volcano. The modest degradation at larger scales (from 2.02× at 64 GPUs to 1.82× at 512 GPUs) arises because the ESP tâtonnement converges more slowly as the number of interacting jobs grows; however, the per-epoch scheduling latency remains well under the 5-second epoch interval even at 512 GPUs (13.7 ms), confirming that the $O(M \cdot I \cdot N \cdot \bar{K})$ complexity analysis in §3.4 translates to practical scalability.

We also examine sensitivity to load intensity by varying the job arrival rate at the 128-GPU scale. At low load (50% of peak arrival rate), all schedulers perform comparably because resources are abundant. As load increases to 100% and 150% of peak, ElastiCo's advantage widens: the JCT improvement over Volcano grows from 1.8× at 50% load to 2.9× at 150% load. Under heavy contention, the combination of RST-enabled reshaping and ESP-driven pricing becomes especially valuable, as it enables the scheduler to pack more jobs onto the same hardware by finding interference-safe configuration combinations that rigid schedulers cannot access.

5.7. System Overhead

We quantify four sources of overhead introduced by ElastiCo.

Profiling overhead. Initial profile generation takes 3–5 minutes per (model architecture, configuration) pair, including warm-up and measurement. However, due to the profile cache (§4.1), the amortized profiling overhead is low: 94% of job arrivals in our evaluation match an existing profile, requiring zero additional profiling time. For the remaining 6%, profiling runs concurrently on a dedicated profiling GPU and does not delay other scheduling decisions.

Scheduling latency. Table 9 breaks down the per-epoch scheduling latency by component. The end-to-end latency is dominated by the ESP tâtonnement (typically 3–5 iterations) and IAC prediction; bin-packing and PDS coordination add minimal overhead. At 64 GPUs with 30 active jobs, the total scheduling latency is 4.2 ms; at 512 GPUs with 200 active jobs, it grows to 13.7 ms–both negligible relative to the 5-second scheduling epoch.

Reconfiguration overhead. Configuration transitions vary in scope and cost. Flag toggles (e.g., enabling prefix caching) take effect immediately via a side-channel RPC to the data-plane agent (sub-millisecond). Engine restarts (e.g., changing continuous-batching parameters in vLLM) require draining in-flight requests and relaunching the serving process (5–15 s). Checkpoint–resume transitions (e.g., switching activation recomputation or precision mode) serialize

Table 9
Per-epoch scheduling latency breakdown (64-GPU testbed, 30 active jobs).

Component	Latency (ms)
ESP tâtonnement (3–5 iterations)	2.1
IAC prediction (all pairs)	1.2
Bin-packing	0.5
PDS coordination	0.4
Total	**4.2**

model state to persistent storage, terminate the pod, and relaunch under the new configuration (15–45 s depending on model size). In the 24-hour evaluation trace, the scheduler triggered an average of 2.3 checkpoint–resume transitions per hour per GPU, with each transition consuming less than 0.3% of total GPU-hours.

Interference monitoring overhead. The TDOP data-plane agent consumes less than 1% additional CPU and negligible GPU overhead, as it relies on passive DCGM sampling (1-second intervals) and lightweight `LD_PRELOAD` wrappers that add no measurable latency to CUDA kernel launches. The memory footprint of the per-process ring buffer is 4 MB.

6. Related Work

Training-inference co-location. Co-locating training and inference workloads on a shared GPU cluster is an emerging strategy for improving cluster utilization. SIRIUS [6] prioritizes inference tasks with unrestricted GPU access and runs training on leftover resources, using millisecond-level gradient-aware memory adjustment and SLO-aware reallocation for fast GPU memory handover, but treats each job's resource configuration as fixed, which limits packing flexibility when cluster-wide resource pressure changes. SMore [7] enhances GPU utilization through serverless-based co-location scheduling with a degradation prediction model, yet focuses exclusively on training-side performance and does not manage offline inference throughput. ConServe [29] co-serves latency-critical online requests with latency-tolerant offline tasks on shared GPUs through fine-grained harvesting–token-level scheduling, layer-wise preemption, and incremental KV cache management–but targets a single LLM serving engine rather than cluster-wide training-inference co-location. GPUColo [8] provides latency-guaranteed co-location through a two-tier mechanism–an outer tier that dynamically adjusts MPS thread percentages via spatial sharing and an inner tier that periodically sleeps training processes for prompt inference latency control–but confines adaptation to GPU-local knobs without cluster-wide multi-resource optimization across both workload classes. Mudi [1] multiplexes inference services with training tasks through spatial sharing, using piece-wise linear profiling to quantify resource interference and adaptive batching to handle dynamic workloads, but its co-location policy optimizes GPU-percentage allocation without reshaping per-job configurations. Llumnix [30] reschedules requests across LLM serving instances at runtime via live migration to improve load balancing, reduce fragmentation, and differentiate priorities, but operates entirely within the inference serving layer without addressing training–inference co-location. LeMix [31] co-locates LLM serving and training on shared multi-GPU nodes, using offline profiling and runtime scheduling to adapt resource allocation based on workload characteristics and co-execution interference; however, it focuses on single-model retraining scenarios and does not address cluster-wide heterogeneous job scheduling. In contrast, ElastiCo jointly optimizes both training and inference configurations through RST, coordinates resource allocation via market-based pricing (ESP), and feeds interference predictions (IAC) back into the allocation loop–capabilities that none of the above systems integrate simultaneously.

Elastic DL cluster scheduling. Several schedulers allow dynamic adaptation of job resource allocations. Pollux [9] co-adaptively optimizes both per-job training hyperparameters (batch size, learning rate) and cluster-wide GPU allocation using a goodput metric that combines system throughput with statistical efficiency; however, it targets training jobs exclusively and does not handle inference workloads or co-location interference. Sia [32] introduces a scalable scheduling formulation that matches elastic resource-adaptive jobs to heterogeneous GPU types and counts, with low-overhead throughput-model bootstrapping and the first support for elastic scaling of hybrid parallel (data + pipeline) jobs, but focuses exclusively on training and does not model co-location interference. Shockwave [33] extends classic market theory to dynamic settings and uses stochastic dynamic programming with future planning to co-optimize fairness and efficiency under dynamic job adaptation; while it shares the market-theoretic spirit with

ElastiCo's ESP module, it does not support intra-GPU co-location or interference-aware placement. Lucid [10] uses lightweight profiling and an indolent packing algorithm to schedule DL training jobs non-intrusively, minimizing average JCT without modifying user code; it targets training workloads exclusively and does not model co-location interference. PowerFlow [34] dynamically allocates GPUs and adjusts power configurations to minimize average JCT under an energy budget, demonstrating energy-efficiency–performance co-optimization in GPU clusters. Mist [35] comprehensively co-optimizes memory footprint reduction techniques (checkpointing, offloading, redundancy elimination) alongside parallelism strategies for LLM training via symbolic performance analysis and imbalance-aware hierarchical tuning, but targets training-only optimization without co-location considerations. Earlier systems such as Gandiva [36] (time-slicing and migration) and Tiresias [37] (information-agnostic priority scheduling that minimizes average JCT for jobs with unpredictable durations) treat job resource requirements as fixed. ElastiCo extends the elastic scheduling paradigm by formalizing configuration flexibility across both training and inference workloads (via RST) and coupling it with a decentralized pricing mechanism that scales to large clusters.

Market-based resource allocation. ElastiCo's ESP module builds on a rich theoretical foundation in market-based resource allocation. Kelly et al. [38] establish proportional fairness and shadow pricing for communication networks, showing that distributed agents responding to price signals can converge to a socially optimal allocation. Dominant Resource Fairness (DRF) [39] extends max-min fairness to multiple resource types, providing a fairness baseline for multi-resource environments. Carbyne [40] applies altruistic resource sharing in multi-tenant clusters. In the GPU scheduling domain, Shockwave [33] extends the Fisher market framework to dynamic settings and solves the allocation via stochastic dynamic programming with future planning, rather than iterative price adjustment. ElastiCo differs in two key aspects: (i) it employs tâtonnement-style iterative price updates (Eq. 8) that naturally accommodate the discrete configuration sets produced by RST, and (ii) it embeds interference penalties from IAC directly into the pricing loop, enabling the market mechanism to account for co-location externalities that conventional market formulations ignore.

GPU sharing and interference modeling. Intra-GPU sharing mechanisms determine *how* multiple workloads can safely share a single GPU. AntMan [41] co-designs cluster scheduling with DL frameworks to dynamically scale GPU memory and computation, enabling opportunistic co-execution of multiple jobs on shared GPUs without interference. Salus [42] enables fine-grained GPU sharing via two primitives–fast job switching and memory sharing–through iteration-level scheduling, and PipeSwitch [43] pipelines model transmission and GPU execution by exploiting the layered structure of neural networks, achieving millisecond-scale task switching for DL application time-sharing. NVIDIA MPS [44] and MIG [45] provide hardware-level spatial sharing primitives. Orion [46] transparently intercepts GPU kernel launches and schedules work at individual operator granularity, accounting for each operator's compute and memory requirements to minimize interference among co-located workloads. More recently, ISACPP [47] builds an edge-fusion gated graph attention network incorporating DL model structures and GPU types to predict co-location performance, and proposes a multi-stage interference quantification model to identify minimum-interference GPU placements. USHER [48] maximizes GPU utilization for ML inference by profiling per-kernel resource requirements, scheduling co-located models with an interference-aware heuristic, and merging operator graphs to reduce cache contention. KACE [49] predicts co-location interference from exclusive kernel-level GPU metrics with minimal profiling overhead, enabling lightweight co-location decisions under CUDA-MPS spatial sharing. Hu et al. [4] present an in-depth characterization of LLM development workloads in GPU datacenters, revealing resource utilization imbalances and the impact of frequent job failures at scale. ElastiCo's IAC module uses a DNN-based regressor over hardware-counter and task-level features, which generalizes across heterogeneous model architectures without requiring computation graph information; furthermore, IAC feeds its predictions back into ESP's pricing loop rather than acting as a standalone post-hoc filter.

LLM inference and serving. A separate line of work optimizes the serving stack for large language models. vLLM [15] introduces PagedAttention for efficient KV-cache memory management. DistServe [50] assigns prefill and decoding to separate GPUs, co-optimizing per-phase resource allocation and parallelism strategies to meet both TTFT and TPOT latency targets. Splitwise [51] splits the compute-intensive prompt computation and memory-intensive token generation phases onto separate machines, enabling phase-specific hardware matching for higher throughput at lower cost. AlpaServe [52] exploits model parallelism not only for scaling large models but also for statistical multiplexing across devices, determining efficient placement and parallelization strategies to reduce serving latency under bursty workloads. SpotServe [53] dynamically adapts LLM parallelization configurations on preemptible cloud instances, using optimal migration planning and stateful inference recovery to maintain serving quality despite frequent instance preemptions. Vidur [54] provides a high-fidelity simulation framework for LLM inference that models

operator performance via profiling and predictive modeling, enabling rapid exploration of parallelization, batching, and scheduling configurations at scale without expensive real-cluster experiments. DeepServe [55] is a serverless AI platform that efficiently serves LLMs at scale in cloud environments, integrating PD-disaggregated and PD-colocated scheduling, NPU-centric execution, and rapid scaling optimizations for production deployment on large Ascend clusters. SGLang [14] introduces RadixAttention for KV cache reuse and compressed finite state machines for structured output decoding, achieving up to 6.4 × higher throughput on complex LLM programs. These systems optimize single-workload inference serving within an individual serving engine and are *complementary* to ElastiCo, which operates at the cluster scheduling layer to determine resource allocation and co-location decisions above the serving stack.

7. Conclusion

This paper presented ElastiCo, an elastic co-location framework for multi-tenant GPU clusters that concurrently serve deep learning training and offline LLM inference workloads. Our central thesis is that cluster schedulers must jointly reason about *intra-job configuration flexibility* and *inter-job interference* to unlock the utilization headroom left by static-allocation policies–and that solving either in isolation yields only marginal gains, because the unaddressed dimension becomes the binding constraint.

Guided by this, we introduced three integrated mechanisms: (1) *Resource Shape Transformation (RST)*, which exposes each job as a family of feasible resource–performance profiles by systematically exploring activation recomputation, micro-batch sizing, mixed-precision modes, and KV-cache configurations–making intra-job configuration flexibility a first-class scheduling dimension for the first time in the training–inference co-location setting; (2) *Elastic Shadow Pricing (ESP)*, which decomposes the resulting combinatorial multi-resource allocation problem into per-job subproblems via Lagrangian relaxation with Pigouvian interference penalties, scaling to hundreds of concurrent jobs in under 15 ms; and (3) *Interference-Aware Co-location (IAC)*, which predicts pairwise slowdown under NVIDIA MPS co-execution using a DNN-based model on hardware-counter and task-level features (7.6% MAPE) and enforces adaptive tolerance thresholds to bound throughput degradation. The *Phase-Aware Disaggregated Scheduling (PDS)* module orchestrates these components in a closed-loop control cycle with queue-aware capacity reservation and runtime adaptation.

Implemented as Kubernetes-native middleware requiring no modifications to user code, ElastiCo reduces average job completion time by up to 2.94×, increases cluster throughput by 2.02×, raises GPU utilization from approximately 25% to 46%, and reduces the additional GPU instances required for concurrent offline inference by 44% compared to static partitioning–all while meeting throughput targets for 98.3% of jobs. Ablation experiments confirm that each component contributes complementary gains: RST provides the largest gain in scheduling flexibility (53% JCT increase when disabled), ESP improves global allocation efficiency through heterogeneity-aware coordination, and IAC is essential for preventing localized performance collapse under aggressive GPU sharing.

Limitations. We identify four limitations that scope the current contribution. *(1) Single-GPU configuration scope:* the current RST implementation and evaluation focus on per-GPU configuration knobs (batch size, checkpointing, precision, KV-cache parameters). Although the RST abstraction naturally extends to multi-GPU parallelism strategies (data, tensor, pipeline parallelism), this extension introduces inter-GPU communication modeling and is left for future work. *(2) Hardware homogeneity:* all experiments use NVIDIA A100-40GB GPUs; profile accuracy on other GPU generations (e.g., H100, L40S) requires re-profiling. *(3) Moderate cluster scale:* physical validation is limited to 64 GPUs, with simulation extending to 512 GPUs. Larger-scale deployments may reveal additional scheduling dynamics. *(4) IAC generalization:* the interference predictor is trained on the 12 workloads in our evaluation mix; accuracy may degrade on model architectures with substantially different hardware utilization profiles (e.g., mixture-of-experts or diffusion models).

Future work. Promising extensions include: (i) integrating multi-GPU parallelism strategies into RST, enabling the scheduler to jointly select parallelism configurations and per-GPU knobs–this would subsume systems like Mist [35] into the RST framework; (ii) supporting heterogeneous multi-generation GPU fleets, where RST profiles must capture hardware-specific performance characteristics; (iii) incorporating network bandwidth and storage I/O into the resource-shape representation for data-intensive distributed training; and (iv) adopting online learning for the IAC predictor to handle novel model architectures without offline profiling, enabling continuous adaptation as the workload mix evolves.

Funding

This work is supported in part by National Key R&D Program of China (Grant No. 2024YFB4505604), in part by the National Natural Science Foundation of China (Grant No. 62402024), in part by the Beijing Natural Science Foundation (Grant No. L241050), in part by the Fundamental Research Funds for the Central Universities, and, last but not least, by Kuaishou Research Fund.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work the author(s) used ChatGPT in order to improve the linguistic clarity and readability of the manuscript. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the published article.

References

[1] W. Chen, C. Lu, H. Xu, K. Ye, C. Xu, Multiplexing dynamic deep learning workloads with SLO-awareness in GPU clusters, in: Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25), 2025, pp. 589–604. URL: https://doi.org/10.1145/3689031.3696074. doi:10.1145/3689031.3696074.

[2] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, Y. Ding, MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters, in: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 2022, pp. 945–960. URL: https://www.usenix.org/conference/nsdi22/presentation/weng.

[3] Y. Gao, Y. He, X. Li, B. Zhao, H. Lin, Y. Liang, J. Zhong, H. Zhang, J. Wang, Y. Zeng, K. Gui, J. Tong, M. Yang, An empirical study on low GPU utilization of deep learning jobs, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), 2024. doi:10.1145/3597503.3639232.

[4] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, T. Zhang, Characterization of large language model development in the datacenter, in: 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), 2024, pp. 709–729. URL: https://www.usenix.org/conference/nsdi24/presentation/hu.

[5] C. Lv, X. Shi, D. Liang, W. Tan, X. Zhao, SpecInF: Exploiting idle GPU resources in distributed DL training via speculative inference filling, in: Network and Parallel Computing: 20th IFIP WG 10.3 International Conference, NPC 2024, Proceedings, Part I, 2025, pp. 146–158. URL: https://doi.org/10.1007/978-981-96-2830-8_12. doi:10.1007/978-981-96-2830-8_12.

[6] J. Wang, Y. Wang, M. Han, R. Chen, Colocating ML inference and training with fast GPU memory handover, in: 2025 USENIX Annual Technical Conference (USENIX ATC 25), 2025, pp. 1657–1675. URL: https://www.usenix.org/conference/atc25/presentation/wang-jiali.

[7] J. Liu, Z. Cai, Y. Liu, H. Li, Z. Zhang, R. Ma, R. Buyya, SMore: Enhancing GPU utilization in deep learning clusters by serverless-based co-location scheduling, IEEE Transactions on Parallel and Distributed Systems 36 (2025) 903–917. doi:10.1109/TPDS.2025.3548320.

[8] G. Chen, S. Subramaniyan, X. Wang, Latency-guaranteed co-location of inference and training for reducing data center expenses, in: Proc. IEEE International Conference on Distributed Computing Systems (ICDCS), 2024, pp. 473–484. doi:10.1109/ICDCS60910.2024.00051.

[9] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, E. P. Xing, Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning, in: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), 2021, pp. 1–18. URL: https://www.usenix.org/conference/osdi21/presentation/qiao.

[10] Q. Hu, M. Zhang, P. Sun, Y. Wen, T. Zhang, Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), 2023, pp. 457–472. doi:10.1145/3575693.3575705.

[11] Y. Lin, S. Wu, S. Luo, H. Xu, H. Shen, C. Ma, M. Shen, L. Chen, C. Xu, L. Qu, K. Ye, Understanding diffusion model serving in production: A top-down analysis of workload, scheduling, and resource efficiency, in: Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC '25), 2025, pp. 1–15. doi:10.1145/3772052.3772206.

[12] T. Chen, B. Xu, C. Zhang, C. Guestrin, Training deep nets with sublinear memory cost, 2016. URL: https://arxiv.org/abs/1604.06174. arXiv:1604.06174, preprint.

[13] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, H. Wu, Mixed precision training, in: International Conference on Learning Representations, 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[14] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, Y. Sheng, SGLang: Efficient execution of structured language model programs, in: Proceedings of the 38th International Conference on Neural Information Processing Systems, NeurIPS '24, Curran Associates Inc., Red Hook, NY, USA, 2024. URL: https://dl.acm.org/doi/10.5555/3737916.3739916.

[15] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, I. Stoica, Efficient memory management for large language model serving with PagedAttention, in: Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23), 2023, pp. 611–626. doi:10.1145/3600006.3613165.

[16] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778. doi:10.1109/CVPR.2016.90.

[17] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Association for Computational Linguistics, Minneapolis, Minnesota, 2019, pp. 4171–4186. URL: https://aclanthology.org/N19-1423/. doi:10.18653/v1/N19-1423.

[18] A. Radford, L. Metz, S. Chintala, Unsupervised representation learning with deep convolutional generative adversarial networks, 2016. URL: https://arxiv.org/abs/1511.06434. arXiv:1511.06434, preprint.

[19] R. Q. Charles, H. Su, M. Kaichun, L. J. Guibas, Pointnet: Deep learning on point sets for 3d classification and segmentation, in: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 77–85. doi:10.1109/CVPR.2017.16.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 30, Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[21] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (1997) 1735–1780. URL: https://doi.org/10.1162/neco.1997.9.8.1735. doi:10.1162/neco.1997.9.8.1735.

[22] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, T.-S. Chua, Neural collaborative filtering, in: Proceedings of the 26th International Conference on World Wide Web, WWW '17, International World Wide Web Conferences Steering Committee, 2017, pp. 173–182. URL: https://doi.org/10.1145/3038912.3052569. doi:10.1145/3038912.3052569.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, 2017. URL: https://arxiv.org/abs/1707.06347. arXiv:1707.06347, preprint.

[24] DeepSeek-AI, :, X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu, H. Gao, K. Gao, W. Gao, R. Ge, K. Guan, D. Guo, J. Guo, G. Hao, Z. Hao, Y. He, W. Hu, P. Huang, E. Li, G. Li, J. Li, Y. Li, Y. K. Li, W. Liang, F. Lin, A. X. Liu, B. Liu, W. Liu, X. Liu, X. Liu, Y. Liu, H. Lu, S. Lu, F. Luo, S. Ma, X. Nie, T. Pei, Y. Piao, J. Qiu, H. Qu, T. Ren, Z. Ren, C. Ruan, Z. Sha, Z. Shao, J. Song, X. Su, J. Sun, Y. Sun, M. Tang, B. Wang, P. Wang, S. Wang, Y. Wang, Y. Wang, T. Wu, Y. Wu, X. Xie, Z. Xie, Z. Xie, Y. Xiong, H. Xu, R. X. Xu, Y. Xu, D. Yang, Y. You, S. Yu, X. Yu, B. Zhang, H. Zhang, L. Zhang, L. Zhang, M. Zhang, M. Zhang, W. Zhang, Y. Zhang, C. Zhao, Y. Zhao, S. Zhou, S. Zhou, Q. Zhu, Y. Zou, Deepseek llm: Scaling open-source language models with longtermism, 2024. URL: https://arxiv.org/abs/2401.02954. arXiv:2401.02954, preprint.

[25] A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang, G. Dong, H. Wei, H. Lin, J. Tang, J. Wang, J. Yang, J. Tu, J. Zhang, J. Ma, J. Yang, J. Xu, J. Zhou, J. Bai, J. He, J. Lin, K. Dang, K. Lu, K. Chen, K. Yang, M. Li, M. Xue, N. Ni, P. Zhang, P. Wang, R. Peng, R. Men, R. Gao, R. Lin, S. Wang, S. Bai, S. Tan, T. Zhu, T. Li, T. Liu, W. Ge, X. Deng, X. Zhou, X. Ren, X. Zhang, X. Wei, X. Ren, X. Liu, Y. Fan, Y. Yao, Y. Zhang, Y. Wan, Y. Chu, Y. Liu, Z. Cui, Z. Zhang, Z. Guo, Z. Fan, Qwen2 technical report, 2024. URL: https://arxiv.org/abs/2407.10671. arXiv:2407.10671, preprint.

[26] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed, Mistral 7b, 2023. URL: https://arxiv.org/abs/2310.06825. arXiv:2310.06825, preprint.

[27] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco, A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis,

I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, Z. Ma, The llama 3 herd of models, 2024. URL: https://arxiv.org/abs/2407.21783. arXiv:2407.21783, preprint.

[28] Volcano Community, Volcano: Cloud native batch computing platform, https://volcano.sh/, 2024. CNCF Incubating Project.

[29] Y. Qiao, S. Anzai, S. Yu, H. Ma, S. Yang, Y. Wang, M. Kim, Y. Wu, Y. Zhou, J. Xing, J. E. Gonzalez, I. Stoica, H. Xu, ConServe: Fine-grained GPU harvesting for LLM online and offline co-serving, 2025. URL: https://arxiv.org/abs/2410.01228. arXiv:2410.01228, preprint.

[30] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, W. Lin, Llumnix: Dynamic scheduling for large language model serving, in: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), 2024, pp. 173–191. URL: https://www.usenix.org/conference/osdi24/presentation/sun-biao.

[31] Y. Li, Z. Li, Y. Zhu, C. Liu, LeMix: Unified scheduling for LLM training and inference on multi-GPU systems, 2025. URL: https://arxiv.org/abs/2507.21276. arXiv:2507.21276, preprint.

[32] S. Jayaram Subramanya, D. Arfeen, S. Lin, A. Qiao, Z. Jia, G. R. Ganger, Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling, in: Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23), 2023, pp. 642–657. URL: https://doi.org/10.1145/3600006.3613175. doi:10.1145/3600006.3613175.

[33] P. Zheng, R. Pan, T. Khan, S. Venkataraman, A. Akella, Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning, in: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), 2023, pp. 703–723. URL: https://www.usenix.org/conference/nsdi23/presentation/zheng.

[34] D. Gu, X. Xie, G. Huang, X. Jin, X. Liu, PowerFlow: Energy-efficient GPU clusters scheduling for deep learning, 2023. URL: https://arxiv.org/abs/2304.06381. arXiv:2304.06381, preprint.

[35] Z. Zhu, C. Giannoula, M. Andoorveedu, Q. Su, K. Mangalam, B. Zheng, G. Pekhimenko, Mist: Efficient distributed training of large language models via memory-parallelism co-optimization, in: Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25), 2025, pp. 1298–1316. URL: https://doi.org/10.1145/3689031.3717461. doi:10.1145/3689031.3717461.

[36] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, L. Zhou, Gandiva: Introspective cluster scheduling for deep learning, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 595–610. URL: https://www.usenix.org/conference/osdi18/presentation/xiao.

[37] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, C. Guo, Tiresias: A GPU cluster manager for distributed deep learning, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19), 2019, pp. 485–500. URL: https://www.usenix.org/conference/nsdi19/presentation/gu.

[38] F. P. Kelly, A. K. Maulloo, D. K. H. Tan, Rate control for communication networks: Shadow prices, proportional fairness and stability, Journal of the Operational Research Society 49 (1998) 237–252. doi:10.1057/palgrave.jors.2600523.

[39] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant resource fairness: Fair allocation of multiple resource types, in: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11), USENIX Association, 2011, pp. 323–336. URL: https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types.

[40] R. Grandl, M. Chowdhury, A. Akella, G. Ananthanarayanan, Altruistic scheduling in multi-resource clusters, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 65–80. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl_altruistic.

[41] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, Y. Jia, AntMan: Dynamic scaling on GPU clusters for deep learning, in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, 2020, pp. 533–548. URL: https://www.usenix.org/conference/osdi20/presentation/xiao.

[42] P. Yu, M. Chowdhury, Fine-grained GPU sharing primitives for deep learning applications, in: Proceedings of Machine Learning and Systems, volume 2, 2020, pp. 98–111. URL: https://proceedings.mlsys.org/paper_files/paper/2020/file/d9cd83bc91b8c36a0c7c0fcca59228f2-Paper.pdf.

[43] Z. Bai, Z. Zhang, Y. Zhu, X. Jin, PipeSwitch: Fast pipelined context switching for deep learning applications, in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 499–514. URL: https://www.usenix.org/conference/osdi20/presentation/bai.

[44] NVIDIA, Multi-Process Service, Technical Report, NVIDIA Corporation, 2019. URL: https://docs.nvidia.com/deploy/mps/index.html.

[45] NVIDIA, Multi-Instance GPU, Technical Report, NVIDIA Corporation, 2020. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[46] F. Strati, X. Ma, A. Klimovic, Orion: Interference-aware, fine-grained GPU sharing for ML applications, in: Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24), 2024, pp. 1075–1092. doi:10.1145/3627703.3629578.

[47] Z. Liu, Y. Cheng, C. Chen, J. Hu, R. Fu, D. Zhang, ISACPP: Interference-aware scheduling approach for deep learning training workloads based on co-location performance prediction, IEEE Transactions on Parallel and Distributed Systems 36 (2025) 1591–1607. doi:10.1109/TPDS.2025.3577796.

[48] S. S. Shubha, H. Shen, A. Iyer, USHER: Holistic interference avoidance for resource optimized ML inference, in: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), 2024, pp. 947–964. URL: https://www.usenix.org/conference/osdi24/presentation/shubha.

[49] B.-S. Han, T. Paul, Z. Liu, A. Gandhi, KACE: Kernel-aware colocation for efficient GPU spatial sharing, in: Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24), 2024, pp. 460–469. URL: https://doi.org/10.1145/3698038.3698555. doi:10.1145/3698038.3698555.

[50] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, H. Zhang, DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving, in: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), 2024, pp. 193–210. URL: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

[51] P. Patel, E. Choukse, C. Zhang, A. Shah, I. n. Goiri, S. Maleki, R. Bianchini, Splitwise: Efficient generative LLM inference using phase splitting, in: 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), 2024, pp. 118–132. doi:10.1109/ISCA59077.2024.00019.

[52] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, I. Stoica, AlpaServe: Statistical multiplexing with model parallelism for deep learning serving, in: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), 2023, pp. 663–679. URL: https://www.usenix.org/conference/osdi23/presentation/li-zhouhan.

[53] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, Z. Jia, SpotServe: Serving generative large language models on preemptible instances, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), 2024, pp. 1112–1127. doi:10.1145/3620665.3640411.

[54] A. Agrawal, N. Kedia, J. Mohan, A. Panwar, N. Kwatra, B. S. Gulavani, R. Ramjee, A. Tumanov, Vidur: A large-scale simulation framework for LLM inference, in: P. Gibbons, G. Pekhimenko, C. D. Sa (Eds.), Proceedings of Machine Learning and Systems, volume 6, 2024, pp. 351–366. URL: https://proceedings.mlsys.org/paper_files/paper/2024/file/b74a8de47d2b3c928360e0a011f48351-Paper-Conference.pdf.

[55] J. Hu, J. Xu, Z. Liu, Y. He, Y. Chen, H. Xu, J. Liu, J. Meng, B. Zhang, S. Wan, G. Dan, Z. Dong, Z. Ren, C. Liu, T. Xie, D. Lin, Q. Zhang, Y. Yu, H. Feng, X. Chen, Y. Shan, DeepServe: Serverless large language model serving at scale, in: 2025 USENIX Annual Technical Conference (USENIX ATC 25), 2025. URL: https://www.usenix.org/conference/atc25/presentation/hu-junhao.